A HIERARCHICAL METHOD OF PERFORMING GLOBAL OPTIMIZATIONS

By

LAURIE A. WHITE

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1990

ACKNOWLEDGEMENTS

I must first thank the wonderful teachers along the way who made this possible for me. From Dorothy Buckley who had me going to the library in the 6th grade to find out more, through Diana Brantley and Sue Sturgill, who actually expected me to think in high school, and Glenn Kesler who frightened me with all that I did not know in a logic tutorial at the University of Virginia (but then went on to help me learn much of it), to those professors at the University of Florida who gave me the background, motivation, and encouragement to undertake this dissertation.

I especially appreciate the work done by my committee chairman, Dr. Gerhard Ritter, and all of the members of my committee. Jeffie Woodham, the graduate secretary of the Computer and Information Sciences Department, made everything much easier by both knowing exactly what I had to do at every step along the way and being ready with a smile as she helped me through.

Special thanks go to Dr. Joseph N. Wilson, my cochairman. He spent literally hundreds of hours working with me in all phases of this project, keeping me excited when things were going well and encouraged when they were not. He was available to me at even the worst times, including a year of meetings at 5 p.m. Fridays.

Thanks go to my mother and my father, who raised me in a home where anything was possible. When things started to seem difficult, it was good to have that to fall back on.

And finally, my greatest thanks go to my husband Charles Engelke, who assures me, that although writing a dissertation is difficult, living with someone writing a dissertation is far worse. Even when I wasn't there for him, he was always there for me, challenging me to do my best, and then some.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Symbol	Meaning	Page
x, y, z, u	Simple integer variable	15
$a[s]$	Array reference	15
v	Integer variable	15
m, n	Integer constant	15
s	Integer expression	15
$\oplus$	Integer binary operator	15
α	Integer	15
V	Set of integers	15
b	Boolean expression	16
$\ominus$	Boolean binary operator	16
β	Truth value	16
W	Set of truth values	16
S	Statement	16
σ	State	16
Σ	Set of states	16
$\mathcal{L}$	Location of a variable	16
ξ	Intermediate variable	16
$\sigma\{\alpha/\xi\}$	State variant	17
sep	Always separate	18
c	Complexity	19
$S[s/x]$	Textual substitution	20
$v < s/x >$	Left hand side substitution	20
$\overline{\alpha}$	Constant with the value of α	26
$\mathcal{R}$	Meaning of an integer expression	26
$\mathcal{W}$	Meaning of a Boolean expression	27
$\mathcal{M}$	Meaning of a statement	28
$=_\sigma$	Equal in state σ	28
$=$	Equal in all states	28

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

A HIERARCHICAL METHOD OF PERFORMING GLOBAL OPTIMIZATIONS

By

Laurie A. White

May 1990

Chairman: Dr. Gerhard X. Ritter
Cochairman: Dr. Joseph N. Wilson
Major Department: Computer and Information Sciences

Program optimization has long been an important function of compilers. Traditionally, global optimizations have been accomplished by collecting large sets of data flow information items about the various statements in the program. This dissertation provides a new approach to global optimizations by introducing a set of provably correct code transformations which can be combined to perform most of the traditional global data flow code optimizations.

First, a small language which could be viewed as an intermediate code of an image processing language is defined syntactically and semantically. Next, a group of primitive source-to-source transformations on this language are described and the cases under which each transformation is valid are proved. Then, these primitive transformations are combined to yield global transformations such as code motion and copy propagation. A new result called loop-conditional joining is also developed from the primitive transformations.

Finally, a prototype system using these techniques is developed. It enables a user to experiment with a variety of code transformations and provides some assistance with heuristics designed to improve code.

CHAPTER 1
INTRODUCTION

While straightforward implementations of the image algebra as a means of specifying image processing algorithms have been successful in providing people a uniform means of discussing these algorithms, these implementations have produced some programs which are highly inefficient in using machine resources. This study was motivated by the desire to find ways to optimize image algebra code. Unfortunately, most of the traditional optimization techniques were ill-suited to the gross inefficiencies introduced by direct translation of image algebra programs. Additionally, the proofs of correctness of global data flow optimizations are based on the flow of the program data, rather than the meaning of the program. This work presents a new approach to code optimization designed to solve both of these problems.

Traditionally, code optimization has been classified as either *peephole*, where small pieces of code could have relatively small changes applied, or *global*, where transformations could be made on a larger scale. Determining when global optimizations can be performed has previously been done by looking at the results of global data flow analysis. Rather than collect the large sets of data required for global data flow analysis, I collect the variables set and used by the execution of each statement. Using only this information, primitive transformations can be performed.

These primitive transformations can be proven to be correct using denotational semantics. Previously, any proof of transformation correctness has been carried out by examining the program flow graph without regard to semantics. These provably correct transformations can then be combined to give many of the same global

transformations as those provided by global data flow analysis. Since the basic transformations of the system are so small, they can easily be rearranged and recombined for the task at hand. Thus, I have developed some new and previously unexploited transformations which are highly beneficial for image processing programs.

I have developed a prototype optimizing system which implements all of the primitive transformations and a number of the global optimizations which can be built from them. This system allows a user to experiment with a variety of combinations of the techniques and demonstrates the power and flexibility of this approach.

The remainder of this dissertation is divided into six chapters. Chapter 2 provides a brief background in traditional code optimization techniques, semantic approaches to code optimization, and image processing.

Chapter 3 provides a small language which could be viewed as a simple intermediate language. This chapter provides the syntactic and denotational semantic definition for the language, along with a discussion of the variables set and used by statements and some preliminary results about the language.

The primitive transformations are presented in Chapter 4. A full proof of the correctness of each transformation is given along with its description. Some of the more beneficial global transformations derivable from these primitive transformations are presented in Chapter 5.

Chapter 6 describes the prototype optimizer developed from these transformations. Along with describing the system and its basic operation, it discusses some of the possible combinations of the transformations and examines how this can improve the execution of a sample program.

Finally, Chapter 7 presents conclusions and suggestions for further work.

CHAPTER 2
BACKGROUND MATERIAL

This dissertation combines work from several diverse areas of computer science. First, there is much work previously done in code optimization. Originally, most optimizations were done at the local or peephole level, to correct problems of one particular compiler or to fine tune for one particular architecture. In the early 1970s, there was great interest in more global optimizations. Both of these types of optimizations are discussed in Section 2.1.

Most of the work done in optimization is based on a graphical view of the program being optimized and computes a variety of information based on the structure of the program graph. Instead, I approach the problem from a semantic view and look at the meanings of statements, rather than their positions in the overall program. Background material on semantics is presented in Section 2.2. Much of Chapter 3 is devoted to presenting a denotational semantic background for the language discussed in this dissertation.

Although the language described here and the transformations are general purpose, this work has been motivated by the desire to improve the running time of image processing programs as implemented in the image algebra. A brief introduction to image processing and the image algebra is given in Section 2.3.

Finally, there is no way to guarantee that a single combination of transformations produces an optimal, or even an improved, program. Instead, heuristics must be developed to actually use these transformations to arrive at a new program with a

shorter execution time. This involves the use of expert system techniques as discussed in Section 2.4.

2.1 Code Optimization

Straightforward translation from a high-level language to machine code almost never produces code as good as that which a human machine language programmer could write for the same task. All of the optimizations discussed in this section are definite code improvements. Some of them also have relaxed rules for application if the compiler writer is willing to take the chance that the code may be degraded in certain instances with the relaxed rules. A code-transformation is considered by Aho et al. [1] to be an optimization if it meets the following conditions:

a. The transformation preserves the meaning of the program.

b. The transformation speeds up the execution of the program. Some transformations may be undertaken to reduce the size of the code produced, but the primary emphasis is on speed.

c. The execution time saved by the transformation is at least as much as the time it takes to perform the transformation.

It is this third condition which has kept many of the transformations in Chapter 4 from being considered as optimizations before this time.

2.1.1 Peephole Optimizations

There are some optimizations of statements that can be made with little knowledge of the code surrounding the statements. These are known as peephole optimizations. Only a few instructions (those in the peephole) need to be examined at a time to apply these measures. Peephole optimizations are described below.

Removal of redundant stores and loads. The final step of one instruction may be a `store` of a value and the first step of the next instruction a `load` of the same value.

If this is the case (and both instructions are in the same block), the `load` instruction would not be necessary. (A `load` followed by a `store` would result in the removal of the `store` instruction.)

Removal of unreachable code. Although not all unreachable code can be determined by peephole optimization, some can be. All code following an unconditional branch and before the next labelled statement is unreachable and can be removed.

Flow-of-Control Optimizations. If the peephole used does not require the statements to be contiguous, jump sequences can be examined and optimized. A jump statement to another jump statement can be replaced by a jump to the final destination. This may result in the removal of the intermediate jump statement.

Algebraic simplification. There are many algebraic identities that may be exploited, but the most common ones (and therefore the most beneficial to optimize for) involve statements of the form $x := x + 0$, $x := x * 1$ and $x := x * 0$. These can be replaced with simple assignments or removed entirely.

Reduction in strength. Operations which are considered expensive, such as multiplication and computing squares, are replaced by equivalent operations using less expensive operators (computing a square may be replaced by a multiplication and multiplication may be replaced by a shift operation).

Use of machine idioms. Different target machines may have different operations implemented. Using these special operations may improve the code.

Details on all of these can be found in Aho et al. [1]. Actual use of these techniques for the languages SIMPL and OS/360 FORTRAN H, is reported by Lowry and Medlock [17] and Zelkowitz and Bail [30]. Although these steps seem simple compared to the optimizations that result from global data flow analysis, many redundant

statements can be produced by the compiler front end, which performs code generation for each statement individually without considering what code the surrounding statements have generated.

2.1.2 Global Data Flow Analysis

Global data flow analysis examines the definitions and uses of variables in a program. While there are other uses for data flow analysis, some of which are discussed by Muchnick and Jones [19], the most important is in performing global program optimizations, discussed in Section 2.1.3. Data flow analysis is most commonly done using elimination methods. Allen presents most of the basic concepts of data flow analysis [2]. Other methods have been developed for determining the same information, but using different algorithms. A good overview of elimination methods of data flow analysis is provided by Ryder and Paull [26].

Data flow analysis is based on simple graph theory. It begins with a control flow graph of the program. From that graph, using one of the techniques discussed in the previous paragraph, one first identifies loops suitable for improvement in the code and then computes information for each statement. This information consists of the items discussed below.

Reaching definitions. For each statement, all of the possible definitions of every variable will be calculated. All of the possible definitions for a use of a particular variable in a statement are collected into a list known as the *use-definition chain*, or *ud-chain*.

Live variables. A variable is considered to be alive at a statement if it could be used somewhere in or after the statement.

Definition-use chain. For each definition of a variable, all of the possible uses of the definition will be listed. This is also known as the *du-chain*.

Available expressions. All of the expressions which have already definitely been computed, with no possible change, are calculated for each statement. This will allow redundant subexpression elimination to occur.

Copy statements. For each statement, all statements of the form $a := b$ which have preceded it and have not had either a or b redefined are collected. These will be used in copy propagation.

2.1.3 Global Data Flow Optimizations

Peephole optimization works only with a few statements at a time. When the additional information provided by data flow analysis is known about a program, additional optimizations are possible. The two most important collections of these are the Allen-Cocke catalogue [3] and the Irvine catalogue [29]. These catalogues view the optimizations at a very high level, giving more of an idea of what can be done rather than how it is done. An overview of these catalogues, giving the traditional global data flow optimizations, is presented by Kennedy [16]. These optimizations are below.

Redundant subexpression elimination. A subexpression, once it is calculated, may not need to be recomputed when it is used again. If there is no possible change to the variables in the subexpression between where it is originally computed and where it is recomputed, a new variable is created and the value of the subexpression is assigned to the new variable. Instead of recomputing the subexpression, the value of the new variable is used. This was first discussed by Cocke [7].

Copy propagation. A copy statement, of the form $A := B$, can be removed and all uses of A replaced with B if there are no definitions of A or B between the copy statement and the uses of A. This will not only eliminate extra copy statements and variables the programmer (or high-level language) may have produced, but will also

eliminate many of the extra copy statements produced by other optimizations. If B is a constant, this procedure is called *constant folding.* Constant values may be substituted into expressions wherever possible and the resulting peephole optimization may reduce entire expressions to constants.

Code motion. Statements in a loop that do not depend on the variables that may change in the loop can be moved outside of the loop. This eliminates multiple executions of a statement that only needs to be executed once.

Strength reduction of induction variables. Induction variables are variables that depend on the loop variable for their values. They are typically given values which are some linear function of the loop control variable. Rather than recompute this function every time the loop control variable changes, it can be computed once at the beginning of the loop and incremented each successive time through the loop. This will replace the computation of an expression with a simpler statement.

Elimination of induction variables. Induction variables may in some cases be replaced by the loop control variable which they depend on. This will remove a variable and possibly allow further optimizations.

Dead code elimination. If the du-chain of a statement contains no entries, the definition in the statement is never used, and therefore the statement can be eliminated.

Procedure integration. The body of a procedure can sometimes be substituted for the procedure call. This has the advantage of reducing procedure call overhead, which is very inefficient in some compilers. It may also allow other optimizations to occur and give more restricted ud-chains and du-chains.

Machine-dependent optimizations. If something is known about the target machine's organization, other optimizations to take advantage of the machine's features can be made. The most common machine-dependent optimizations are listed below.

Register allocation. Different machines have different numbers of registers and different types of special-purpose registers. They also have different register manipulation instructions, such as auto-increment. If the optimizer knows about the specifics of the registers, it can better allocate registers to avoid redundant store and load operations and to use these specialized instructions. There is also some optimization which can be done without knowing all of the details of a specific machine. This global machine-independent register allocation utilizes usage counts to determine which values should reside in a limited number of registers and is discussed by Chow [6].

Detection of parallelism. Any instruction which can be coded as a vector operation should be identified if the target machine is a vector machine. Methods to do this are discussed by Schneck [27].

A good overview of the rules for performing subexpression elimination, copy propagation, code motion, and strength reduction can be found in many introductory compiler texts [1,5]. Specific work in the implementation of copy propagation, dead code elimination, code motion, strength reduction and elimination of induction variables and register allocation has been done by Chow using U-Code (described in Section 2.3) as the intermediate language [6].

2.2 Program Semantics

While there has been previous work on the mathematical background of the correctness of program optimization (including an entire book devoted to the subject [28]), this has not included a formal notion of the semantics of the program being optimized. Rosen introduces a high-level approach to the problem, but does not use any sort of semantic definition of the language with which he is working [25]. All of

these works have been based on an informal approach to what program constructs mean.

Cousot presents some of the earliest formal work in this area [8]. He uses an operational semantic framework in which to perform program analyses. I assert that an operationally based framework does not yield the coherent hierarchical framework that denotational semantics provides. This view is shared by a number of others [12,9]. In addition, the current popularity of denotational definitions certainly make optimization work based upon them more appealing. Donzeau-Gouge has explored the application of denotational semantics to program optimization [10]. She demonstrates the applicability of this technique to such optimizations as constant propagation, common subexpression determination, and invariant determination, but does not discuss the elimination of these common subexpressions and invariant expressions. In addition, optimizations such as code motion, loop rolling and unrolling, etc, are not discussed.

2.3 Image Processing and the Image Algebra

Image processing has two main goals. First, images may be processed to enhance them for human use. Image processing is crucial in the images of planets sent back by space probes, for example. Second, images may be processed for machine interpretation. Current work in computer vision includes medical diagnosis, military target acquisition, robotics navigation, and face recognition for television rating services. Introductions to image processing in general can be found in a number of texts [4,13].

Digital images consist of some underlying system of discrete points, called *pixels*, short for picture elements, each of which has a corresponding value in the image. This corresponding value may be some brightness indicator or infrared reading for

the point, or even a vector of values. A common image, the black and white photograph from newspapers, has a 2-dimensional grid of pixels and gray levels indicating the relative brightness or darkness as the values. Some common image processing techniques include detecting edges, smoothing and sharpening, and locating features in an image.

The *AFATL Image Algebra* was developed to provide a standard mathematical environment for image processing. It can perform any gray level image-to-image transformation and has the advantage of having a formal mathematical basis.

The most basic operand in the image algebra is the *image*. Images can have many different coordinate sets and values. A coordinate set (usually denoted $\mathbf{X}$) must be a compact subset of R^n, with n most often being 2, to indicate 2-dimensional images. The image value set (usually denoted $\mathbf{F}$) must be a groupoid. The most common image value sets are integers, natural numbers, real numbers and vectors of integers, natural, or real numbers. An $\mathbf{F}$ valued image, $\mathbf{a}$, on a set of image coordinates, $\mathbf{X}$, is defined to be the graph of the function $a : \mathbf{X} \rightarrow \mathbf{F}$, or:

$$\mathbf{a} = \{(\mathbf{x}, a(\mathbf{x})) : \mathbf{x} \in \mathbf{X}, a(\mathbf{x}) \in \mathbf{F}\}$$

The image algebra provides numerous types of image functions. Binary operations between images include $+$, $-$, $*$, $\wedge$, and $\vee$. These functions will operate pointwise on two images with the same coordinate system. There are also elementary functions, such as the characteristic function (χ) and the sum (Σ) of an image. The characteristic function of an image will be a binary image (that is, an image consisting of just 0 and 1) which is 1 where the pixel meets certain requirements and 0 everywhere else. Thus the characteristic function $\chi_{\leq 7}(A)$ will be 1 where the original image $\mathbf{a}$ has a gray level less than or equal to 7 and will be 0 everywhere else. The sum of an image is defined to be $\sum_{x \in X} a(x)$. There are also a variety of template operations,

both with images and with other templates. These typically require subroutines to implement and are outside the scope of this research. A fuller introduction to the image algebra is presented by Ritter and Wilson [23].

Currently the image algebra is implemented as an extension to FORTRAN, and FORTRAN programs can be written which are converted to standard FORTRAN programs by the Image Algebra FORTRAN preprocessor. A description of Image Algebra FORTRAN is provided by Ritter et al. [24]. Work is underway to implement the image algebra with Image Algebra-C. This work was begun by Perry [22]. Because so much of the image algebra has the potential for vectorization and because of the interest in parallel architectures for image processing in general [11], the intermediate language U-Code [20] was enhanced to include vector instructions. (The addition of vector instructions to a language is discussed by Zosel [31].) The resulting V-Code serves as the intermediate language of Image Algebra-C.

While the image algebra provides powerful notation, these previous attempts at straightforward translation from image algebra programs to lower-level languages have led to highly inefficient code. (This is not just a shortcoming of the image algebra or these implementations. Inefficient translation of code has been a problem for almost as long as there has been translation of code.) Hence, optimizations are important to implement for the image algebra. Inspection of existing optimization techniques showed they were lacking for some of the high-level inefficiencies introduced by the image algebra. Several new or previously unexploited techniques, such as backward copy propagation and loop-conditional joining, are needed to better improve the code.

2.4 Expert Systems

Although this dissertation is not intended to contribute to the field of artificial intelligence, the techniques employed in building simple expert systems proved quite useful in the work presented in Section 6.4. Expert systems are programs which behave as a human expert would. They are particularly useful in situations where no algorithmic solution is possible. An overview of expert systems is presented by Hayes-Roth et al. [14]. For this particular expert system, the simpler reasoning techniques of MACSYMA, as discussed by Martin and Fateman [18], were sufficient.

CHAPTER 3
LANGUAGE DEFINITION

This chapter introduces the terminology used in this dissertation. A small language is defined and its denotational semantics are presented in the style of de Bakker [9]. The language provides both simple and indexed integer variables, integer and boolean expressions, the basic structured statements, an empty statement, and an assignment statement. This is intentionally a simple language. However, it suffices for the ideas presented here. If it were more complex, the proofs in Chapters 4 and 5 would be much more involved, with few, if any, benefits. This simplification of a language to make optimization easier is not without precedent. Rosen discusses movement of optimization decisions from compile time to design time [25].

In keeping with this desire for simplicity, the language is restricted to programs containing loops with bounds fixed at the time of loop entry and does not support subprograms. These language constructs, though amenable to the kind of treatment given other constructs presented here, introduce a level of complexity which would greatly complicate the proofs presented with little or no benefit to the image processing programs being considered.

The first section describes the syntax of the language. The second section discusses variable locations and the states that assign them meanings. Section 3.3 defines a basic complexity measure for expressions and statements in this language, which will be used by some proofs in later sections. The syntax of substitution is given in Section 3.4. Sections 3.5, 3.6 and 3.7 give the semantics of the language, state variants, and substitution. The way statements affect and are affected by the values

Table 3.1. Syntactic Notation

Name	Description	Typical elements
Icon	Integer constants	m, n
Svar	Simple variables	x, y, z, u
Avar	Array variables	a
Ivar	Any integer variable (Svar ∪ Avar)	v, w
Iexp	Integer expressions	s
Bexp	Boolean expressions	b
Stat	Statements	S
	Textual substitution into a member of Iexp (Bexp, Stat)	$s[s_1/y]$
	Left-hand-side substitution into a member of Iexp	$v <v_1/y>$

stored at locations is discussed in Section 3.8 and a static approximation of this is provided in Section 3.9.

3.1 Language Syntax

A brief description of the notation used in this language is given in Table 3.1. Variables in this language are members of the set *Ivar* and may be either simple integer-valued variables (members of the set *Svar*, x, y, x_1, etc.) or integer-expression-indexed integer arrays (members of the set *Avar*, a, a_1, etc.).

Definition 3.1.1 (Integer variables)

$v ::= x \mid a[s]$.

Integer expressions may consist of variables, constants, binary operations and conditional expressions. The set *Icon* will contain all integer constants (m, n, m_1, etc.) while *Iexp* contains the integer expressions (s, s_1, etc.). Actual integer values (α, α_1, etc.) are members of the set *V*.

Definition 3.1.2 (Integer expressions)

$s ::= v \mid m \mid s_1 \oplus s_2 \mid$ **if** b **then** s_1 **else** s_2 **fi**.

(where $\oplus$ is any binary operator).

Boolean expressions may consist of the boolean constants **true** and **false**, relational operations, negation, implication, and subrange inclusion. Boolean expressions (b, b_1, etc.) are in the set *Bexp* and map to truth values (β, β_1, etc.) in the set of truth values *W*.

Definition 3.1.3 (Boolean expressions)

$b ::=$ **true** | **false** | $s_1 \ominus s_2$ | $\neg b$ | s **in** $(s_1 \ldots s_2)$.

(where $\ominus$ is any relational operator).

Statements (members of the set *Stat*, S, S_1, etc.) consist of assignment and empty statements, along with the standard structured constructs of concatenation, selection and iteration. A program in this language will be the same as a statement.

Definition 3.1.4 (Statements)

$S ::= v := s$ | $S_1; S_2$ | **if** b **then** S_1 **else** S_2 **fi** | D | **for** $x := s_1$ **to** s_2 **do** S **od**

(where x does not appear anywhere else).

This language provides no boolean binary operators. If **and** is needed, the statement **if** b_1 **and** b_2 **then** S_1 **else** S_2 **fi** can be replaced with **if** b_1 **then if** b_2 **then** S_1 **else** S_2 **fi else** S_2 **fi**. Similar replacement is possible for the **or** operator. This provides conditional evaluation of the **and** statement, where if the first clause, b_1, is not true, the second clause will not even be evaluated. A number of high-level languages, such as C and Modula-2, have similar conditional evaluation.

3.2 Variable Locations and States

The set *LocV* contains all possible variable locations. A state is a function σ:LocV$\rightarrow$V, mapping locations of variables into integer values. The set of all states is Σ. The location of a variable in a state σ, $\mathcal{L}(v)(\sigma)$, is an intermediate variable (ξ, ξ_1,

etc.). For simple integer variables, the location of the variable is simply the variable itself. With array variables, the location of the variable is given by the array and an integer, the meaning of the index for a particular state. (Definition 3.5.1 explains the meaning of expressions in a given state, $\mathcal{R}$.)

Definition 3.2.1 (Location of a variable)

$$\mathcal{L}(v)(\sigma) = \begin{cases} x & \text{if } v = x \in Svar \\ <a, \mathcal{R}(s)(\sigma)> & \text{if } v = a[s] \in Avar \end{cases}$$

For any program, the domain of σ, a subset of LocV containing all of the intermediate variables of the program assigned values by σ, is assumed to exist. Since there is no need to declare variables in a program, there will be no error handling due to undeclared variables or out-of-bound indices. While these are important considerations in practice, they are not important to the goals of this research.

A state, σ', that assigns the same value to all but one location as another state, σ, is known as a *state variant*. State variants will be important in defining the meaning of statements.

Definition 3.2.2 (Variants of a state)

For each $\sigma \in \Sigma$ and $\alpha \in V$ we write $\sigma\{\alpha/\xi\}$ for each element of Σ which satisfies, for each $\xi_1 \in LocV$:

$$\sigma\{\alpha/\xi\}(\xi_1) = \begin{cases} \alpha & \text{if } \xi_1 = \xi \\ \sigma(\xi_1) & \text{otherwise} \end{cases}$$

Aliasing (two or more names for the same memory location) can cause special problems when determining if a transformation is valid. A transformation which may seem to preserve the meaning of a statement (such as interchanging $x := 7; y := 8$) may in fact change the meaning if there is aliasing (in this case, if x and y refer to the same memory location). Although this language is simple and does not include many of the features which introduce aliases (such as pointers and variable parameters),

because arrays are allowed, it is possible that $a[s_1]$ and $a[s_2]$ refer to the same element of the array a. To avoid possible aliasing problems, the concept of two variables being *always separate* is used.

Definition 3.2.3 (Always separate)

If v_1 *and* $v_2 \in Ivar$, *then we say* v_1 *and* v_2 *are* always separate, *written* $\text{sep}(v_1, v_2)$, *if there is no* $\sigma \in \Sigma$ *with* $\mathcal{L}(v_1)(\sigma) \equiv \mathcal{L}(v_2)(\sigma)$. *We can say* v *is always separate from a set of variables,* I, *written* $\text{sep}(v, I)$, *if,* $\forall v_1 \in I$, $sep(v, v_1)$.

Changing a variable's value may also change its location, as may happen with $a[a[x]]$ when both x and $a[x]$ have the same value. Variables of this form are the exception to many of the following transformations and are said to be *self-referencing.*

Definition 3.2.4 (Strictly non-self-referencing variables)

A variable v *is* strictly non-self-referencing *if* $\mathcal{L}(v)(\sigma\{\alpha/\mathcal{L}(v)\}) = \mathcal{L}(v)(\sigma)$, *for any state* σ *and any integer* α.

Any simple variable is strictly non-self-referencing, as is any indexed variable whose subscript does not include a reference to the array being indexed.

Another form of variable interdependence which may cause trouble occurs when changing the value of one variable may change the location of another. The variable whose location is changed is said to be *location-dependent* on the variable whose value changes.

Definition 3.2.5 (Location-independent)

A variable v *is* location-independent *of a variable* w *if* $\mathcal{L}(v)(\sigma) = \mathcal{L}(v)(\sigma\{\alpha/\mathcal{L}(w)(\sigma)\})$ *for all integers* α *and all states* σ.

Any simple variable is location-independent of any other variable. An array reference is location-independent of another variable if the other variable is not in the expression indexing the array. It should be noted that location independence is not a symmetric relation. The variable x is location-independent of $a[x]$, but $a[x]$ is not location-independent of x.

3.3 Complexity Measures

For the sake of inductive proofs, a structural complexity function c is introduced for the language elements. It is necessary that the complexity of any combination of elements have a complexity greater than the elements comprising it.

Definition 3.3.1 (Complexity of Iexp)

$c(x) = 1$

$c(a[s]) = 1 + c(s)$

$c(m) = 1$

$c(s_1 \oplus s_2) = 1 + c(s_1) + c(s_2)$

$c(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}) = 1 + c(b) + c(s_1) + c(s_2)$

Definition 3.3.2 (Complexity of Bexp)

$c(\texttt{true}) = 1$

$c(\texttt{false}) = 1$

$c(s_1 \ominus s_2) = 1 + c(s_1) + c(s_2)$

$c(\neg b) = 1 + c(b)$

$c(s \texttt{ in } (s_1 \ldots s_2)) = 1 + c(s) + c(s_1) + c(s_2)$

Definition 3.3.3 (Complexity of Stat)

$c(v := s) = 1$

$c(S_1; S_2) = c(S_1) + c(S_2)$

$c(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) = 1 + c\ (b) + c(S_1) + c(S_2)$

$c(D) = 1$

$c(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ do } S \texttt{ od}) = 1 + c(s_1) + c(s_2) + c(S)$

3.4 Syntax of Substitution

Substitution of expressions for simple variables is important in defining the semantics of loops. It will also play a part in statement interchange and absorption. There are two kinds of substitution. Square brackets ([ ]) are used to denote textual substitution into an expression or statement and angle brackets (< >) denote textual substitution into the left-hand-side of an assignment statement.

Substitution for array variables is not defined here for a variety of reasons. The definition of substitution for an array variable traditionally includes conditional expressions, which would greatly complicate the proofs in Chapter 4. Substitution for array variables in statements is even more complicated. This added complexity would provide little new functionality to the language since loop control variables must be simple variables. (I am not alone in this exclusion of substitution for array variables; de Bakker knowingly omits an explanation of $S[v_1/v_2]$ as well [9].)

Definition 3.4.1 (Substitution of expressions into Iexp)

$$x[s/y] \equiv \begin{cases} s & \text{if } y \equiv x \\ x & \text{otherwise} \end{cases}$$

$a[s_1][s/y] \equiv a[s_1[s/y]]$

$m[s/y] \equiv m$

$(s_1 \oplus s_2)[s/y] \equiv (s_1[s/y] \oplus s_2[s/y])$

(**if** b **then** s_1 **else** s_2 **fi**)$[s/y] \equiv$ (**if** $b[s/y]$ **then** $s_1[s/y]$ **else** $s_2[s/y]$ **fi**)

Definition 3.4.2 (Substitution of expressions into Bexp)

true$[s/y] \equiv$ **true**

false$[s/y] \equiv$ **false**

$(s_1 \ominus s_2)[s/y] \equiv (s_1[s/y] \ominus s_2[s/y])$

$(\neg b)[s/y] \equiv \neg(b[s/y])$

(s **in** $(s_1 \ldots s_2)$) $[s/y] \equiv$ ($s[s/y]$ **in** $(s_1[s/y] \ldots s_2[s/y])$)

Left-hand-side substitution will not substitute for a simple variable, but will substitute in the indexing expression of an array reference. Thus $a[x] < s/x > = a[s]$ and $x[s/x] = s$, but $x < s/x > = x$.

Definition 3.4.3 (Left-hand-side substitution)

$$v < s/y > \equiv \begin{cases} x & \text{if } v \equiv x \\ a[s_1[s/y]] & \text{if } v \equiv a[s_1] \end{cases}$$

Substituting into a statement involves substituting for all variables in the statement, much like substituting in an expression. The difference here is that left-hand-side substitution is performed on the left-hand-side of assignment statements and textual substitution is done everywhere else.

Definition 3.4.4 (Substitution of expressions into Stat)

$(v := s_1)[s/y] \equiv v < s/y > := s_1\ [s/y]$

$(S_1; S_2)[s/y] \equiv S_1[s/y];\ S_2[s/y]$

(**if** b **then** S_1 **else** S_2 **fi**)$[s/y] \equiv$ **if** $b[s/y]$ **then** $S_1[s/y]$ **else** $S_2[s/y]$ **fi**

$(D)[s/y] \equiv D$

$(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ do } S \texttt{ od})[s/y] \equiv (\texttt{for } x := s_1[s/y] \texttt{ to } s_2[s/y] \texttt{ do } S[s/y] \texttt{ od})$

A statement with a pair of substitutions (i.e., $S[x/y][n/x]$) may simplify to a statement with a single substitution (i.e., $S[n/y]$) in some cases. This result will be used in other proofs in later chapters (most notably the proof of loop joining in Theorem 5.1.1) and is presented here as an example of a complete inductive proof in this language. In later inductive proofs, only the basis cases will be proven because of the length of the proofs and the fact that there is very little of interest to be found in the inductive steps.

Since statements use both integer and boolean expressions, this result must first be shown for expressions. It is provided in the lemma below. This lemma refers to ivar, the set of all variables in a statement or expression. A fuller definition of ivar is given in Section 3.9.

Lemma 3.4.1

$\models s[n/y] = s[x/y][n/x]$ provided: $x \notin ivar(s)$

and

$\models b[n/y] = b[x/y][n/x]$ provided: $x \notin ivar(b)$

Proof: By simultaneous induction on the complexity of s and b.

Basis: $c(s) = 1$ and $c(b) = 1$

Case 1: $s = y, x \neq y$

$y[n/y] = n$	(Def. of subst. into expr. [3.4.1])
$= x[n/x]$	(Def. of subst. into expr. [3.4.1])
$= (y[x/y])[n/x]$	(Def. of subst. into expr. [3.4.1])

Case 2: $s = z, z \neq y$

$z[n/y] = z$	(Def. of subst. into expr. [3.4.1])
$= z[n/x]$	(Def. of subst. into expr. [3.4.1])
$= (z[x/y])[n/x]$	(Def. of subst. into expr. [3.4.1])

Case 3: $s = m$

$m[n/y] = m$ (Def. of subst. into expr. [3.4.1])
$= m[n/x]$ (Def. of subst. into expr. [3.4.1])
$= (m[x/y])[n/x]$ (Def. of subst. into expr. [3.4.1])

Case 4: $b = \texttt{true}$

$\texttt{true}[n/y] = \texttt{true}$ (Def. of subst. into expr. [3.4.2])
$= \texttt{true}[n/x]$ (Def. of subst. into expr. [3.4.2])
$= (\texttt{true}[x/y])[n/x]$ (Def. of subst. into expr. [3.4.2])

Case 5: $b = \texttt{false}$

$\texttt{false}[n/y] = \texttt{false}$ (Def. of subst. into expr. [3.4.2])
$= \texttt{false}[n/x]$ (Def. of subst. into expr. [3.4.2])
$= (\texttt{false}[x/y])[n/x]$ (Def. of subst. into expr. [3.4.2])

Induction step: Assume that $s[n/y] = s[x/y][n/x]$ and $b[n/y] = b[x/y][n/x]$ whenever $c(s) \leq \mathrm{k}$ and $c(b) \leq \mathrm{k}$ ($\mathrm{k} \geq 0$).

Show that it is true when $c(s) = \mathrm{k} + 1$ and $c(b) = \mathrm{k} + 1$.

Case 1: $s = a[s_1]$

$a[s_1][n/y] = a[s_1[n/y]]$ (Def. of subst. into expr. [3.4.1])
$= a[s_1[x/y][n/x]]$ (Induction hypothesis)
$= a[s_1[x/y]][n/x]$ (Def. of subst. into expr. [3.4.1])
$= a[s_1][x/y][n/x]$ (Def. of subst. into expr. [3.4.1])

Case 2: $s = s_1 \oplus s_2$

$(s_1 \oplus s_2)[n/y] = (s_1[n/y] \oplus s_2[n/y])$ (Def. of subst. into expr. [3.4.1])
$= (s_1[x/y][n/x] \oplus s_2[x/y][n/x])$ (Induction hypothesis)
$= (s_1[x/y] \oplus s_2[x/y])[n/x]$ (Def. of subst. into expr. [3.4.1])
$= (s_1 \oplus s_2)[x/y][n/x]$ (Def. of subst. into expr. [3.4.1])

Case 3: $s = \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}$

$(\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi})[n/y] = \texttt{if}\ b[n/y]\ \texttt{then}\ s_1[n/y]\ \texttt{else}\ s_2[n/y]\ \texttt{fi}$
(Def. of subst. into expr. [3.4.1])
$= \texttt{if}\ b[x/y][n/x]\ \texttt{then}\ s_1[x/y][n/x]\ \texttt{else}\ s_2[x/y][n/x]\ \texttt{fi}$
(Induction hypothesis)
$= (\texttt{if}\ b[x/y]\ \texttt{then}\ s_1[x/y]\ \texttt{else}\ s_2[x/y]\ \texttt{fi})[n/x]$
(Def. of subst. into expr. [3.4.1])
$= (\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi})[x/y][n/x]$ (Def. of subst. into expr. [3.4.1])

Case 4: $b = s_1 \ominus s_2$

$(s_1 \ominus s_2)[n/y] = s_1[n/y] \ominus s_2[n/y]$ (Def. of subst. into expr. [3.4.2])
$= s_1[x/y][n/x] \ominus s_2[x/y][n/x]$ (Induction hypothesis)
$= (s_1[x/y] \ominus s_2[x/y])[n/x]$ (Def. of subst. into expr. [3.4.2])
$= (s_1 \ominus s_2)[x/y][n/x]$ (Def. of subst. into expr. [3.4.2])

Case 5: $b = \neg b$

$(\neg b)[n/y] = \neg(b[n/y])$ (Def. of subst. into expr. [3.4.2])
$= \neg(b[x/y][n/x])$ (Induction hypothesis)
$= (\neg(b[x/y]))[n/x]$ (Def. of subst. into expr. [3.4.2])
$= (\neg b)[x/y][n/x]$ (Def. of subst. into expr. [3.4.2])

Case 6: $b = (s \texttt{ in } (s_1 \ldots s_2))$

$(s \texttt{ in } (s_1 \ldots s_2))[n/y] =$
$= (s[n/y] \texttt{ in } (s_1[n/y] \ldots s_2[n/y]))$ (Def. of subst. into expr. [3.4.2])
$= (s[x/y][n/x] \texttt{ in } (s_1[x/y][n/x] \ldots s_2[x/y][n/x]))$
(Induction hypothesis)
$= (s[x/y] \texttt{ in } (s_1[x/y] \ldots s_2[x/y]))[n/x]$ (Def. of subst. into expr. [3.4.2])
$= (s \texttt{ in } (s_1 \ldots s_2))[x/y][n/x]$ (Def. of subst. into expr. [3.4.2])

□

With the preliminary result proved above, the following lemma can now be proved.

Lemma 3.4.2

$\models S[n/y] = (S[x/y])[n/x]$

Provided:

$x \notin ivar(S)$

Proof: By mathematical induction on the complexity of S.

Basis: $c(S) = 1$
Case 1: $S = D$
$D[n/y] = D$ (Def. of subst. into stat.[3.4.4])
$= D[n/x]$ (Def. of subst. into stat.[3.4.4])
$= D[x/y][n/x]$ (Def. of subst. into stat.[3.4.4])

Case 2: $S = x := s$

$(x := s)[n/y] = x < n/y >:= s[n/y]$ (Def. of subst. into stat.[3.4.4])
$= x := s[n/y]$ (Def. of subst. into l.h.s.[3.4.3])
$= x := s[x/y][n/x]$ (Previous result [3.4.1]))
$= x < n/x >:= s[x/y][n/x]$ (Def. of subst. into l.h.s.[3.4.3])
$= x < x/y >< n/x >:= s[x/y][n/x]$ (Def. of subst. into l.h.s.[3.4.3])
$= (x < x/y >:= s[x/y])[n/x]$ (Def. of subst. into stat.[3.4.4])
$= (x := s)[x/y][n/x]$ (Def. of subst. into stat.[3.4.4])

Case 3: $S = a[s_1] := s$

$(a[s_1] := s)[n/y] = a[s_1] < n/y >:= s[n/y]$ (Def. of subst. into stat.[3.4.4])
$= a[s_1[n/y]] := s[n/y]$ (Def. of subst. into l.h.s.[3.4.3])
$= a[s_1[x/y][n/x]] := s[x/y][n/x]$ (Previous result [3.4.1]))
$= a[s_1[x/y]] < n/x >:= s[x/y][n/x]$ (Def. of subst. into l.h.s.[3.4.3])
$= (a[s_1[x/y]] := s[x/y])[n/x]$ (Def. of subst. into stat.[3.4.4])
$= (a[s_1] < x/y >:= s[x/y])[n/x]$ (Def. of subst. into l.h.s.[3.4.3])
$= (a[s_1] := s)[x/y][n/x]$ (Def. of subst. into stat.[3.4.4])

Induction step: Assume that $S[n/y] = S[x/y][n/x]$. whenever $c(S) \leq k$.

Show it is true when $c(S) = k + 1$.

Case 1: $S = S_1; S_2$

$(S_1; S_2)[n/y] = S_1[n/y]; S_2[n/y]$ (Def. of subst. into stat.[3.4.4])
$= S_1[x/y][n/x]; S_2[x/y][n/x]$ (Induction hypothesis)
$= (S_1[x/y]; S_2[x/y])[n/x]$ (Def. of subst. into stat.[3.4.4])
$= (S_1; S_2)[x/y][n/x]$ (Def. of subst. into stat.[3.4.4])

Case 2: $S =$ **if** b **then** S_1 **else** S_2 **fi**

(**if** b **then** S_1 **else** S_2 **fi**)[n/y] = **if** $b[n/y]$ **then** $S_1[n/y]$ **else** $S_2[n/y]$ **fi**
(Def. of subst. into stat.[3.4.4])
= **if** $b[x/y][n/x]$ **then** $S_1[x/y][n/x]$ **else** $S_2[x/y][n/x]$ **fi**
(Induction hypothesis)
= (**if** $b[x/y]$ **then** $S_1[x/y]$ **else** $S_2[x/y]$ **fi**)$[n/x]$
(Def. of subst. into stat.[3.4.4])
= (**if** b **then** S_1 **else** S_2 **fi**)$[x/y][n/x]$
(Def. of subst. into stat.[3.4.4])

Table 3.2. Semantic Notation

Name	Description	Typical elements
V	Integers	α
W	Truth values (**T** and **F**)	β
Σ	States (functions from LocV to V)	σ
LocV	Intermediate variables (Svar $\cup$ (Avar $\times$ V))	ξ
	Location of Ivar ($\mathcal{L}$: Ivar $\rightarrow$ ($\Sigma \rightarrow$ LocV))	$\mathcal{L}$
	Value of Iexp ($\mathcal{R}$: Iexp $\rightarrow$ ($\Sigma \rightarrow$ V))	$\mathcal{R}$
	Value of Bexp ($\mathcal{W}$: Bexp $\rightarrow$ ($\Sigma \rightarrow$ W))	$\mathcal{W}$
	Value of Stat ($\mathcal{M}$: Stat $\rightarrow$ ($\Sigma \rightarrow \Sigma$))	$\mathcal{M}$
	Variant of a state	$\sigma\{\alpha/\xi\}$
	Constant representing the value of the integer α	$\overline{\alpha}$

Case 3: $S =$ `for` $x :=$ s_1 `to` s_2 `do` S `od`

(`for` $x :=$ m_1 `to` m_2 `do` S `od`)[n/y] = `for` $x :=$ $s_1[n/y]$ `to` $s_2[n/y]$ `do` $S[n/y]$ `od`
(Def. of subst. into stat.[3.4.4])

= `for` $x :=$ $s_1[x/y][n/x]$ `to` $s_2[x/y][n/x]$ `do` $S[x/y][n/x]$ `od`
(Induction hypothesis)

= (`for` $x :=$ $s_1[x/y]$ `to` $s_2[x/y]$ `do` $S[x/y]$ `od`)[n/x]
(Def. of subst. into stat.[3.4.4])

= (`for` $x :=$ s_1 `to` s_2 `do` S `od`)[x/y][n/x] (Def. of subst. into stat.[3.4.4])

□

3.5 Semantics of the Language

Once the syntax of the language, states and substitution is defined, the semantics follow. The notation used in defining the semantics of this language is given in Table 3.2. The meanings of integer and boolean expressions and statements are fairly straightforward. A bar over an integer's value, $\overline{\alpha}$, will be used to indicate a constant with the value of that integer.

Definition 3.5.1 (Semantics of Iexp)

$\mathcal{R}(v)(\sigma) = \sigma(\mathcal{L}(v)(\sigma))$

$\mathcal{R}(m)(\sigma) = \alpha$, *where* α *is the mathematical constant associated with* m

$\mathcal{R}(s_1 \oplus s_2)(\sigma) = \mathcal{R}(s_1)(\sigma) \oplus \mathcal{R}(s_2)(\sigma)$

$\mathcal{R}$(`if` b `then` s_1 `else` s_2 `fi`)$(\sigma) =$ *if* $\mathcal{W}(b)(\sigma)$ *then* $\mathcal{R}(s_1)(\sigma)$ *else* $\mathcal{R}(s_2)(\sigma)$ *fi*

It is further required that all expressions evaluate without error. While there could be special cases for statements which could not be evaluated (such as $x/0$), this only complicates the presentation without providing greater understanding of the correctness proofs here. Thus it is assumed no semantic errors will occur during program execution and no attempt is made to provide meanings for erroneous states or conditions.

Definition 3.5.2 (Semantics of Bexp)

$\mathcal{W}(\texttt{true})(\sigma) = \mathbf{T}$

$\mathcal{W}(\texttt{false})(\sigma) = \mathbf{F}$

$\mathcal{W}(s_1 \ominus s_2)(\sigma) = (\mathcal{R}(s_1)(\sigma) \ominus \mathcal{R}(s_2)(\sigma))$

$\mathcal{W}(\neg b)(\sigma) = \neg \mathcal{W}(b)(\sigma)$

$\mathcal{W}(s \texttt{ in } (s_1 \ldots s_2)) = \mathcal{R}(s)(\sigma) \geq \mathcal{R}(s_1)(\sigma)$ *and* $\mathcal{R}(s)(\sigma) \leq \mathcal{R}(s_2)(\sigma)$

The semantics of statements is also unsurprising. The meaning of a statement in a state σ is just a variant of σ. So, for any statement, $\mathcal{M}(S)(\sigma) = \sigma\{\alpha_1/\xi_1\} \ldots \{\alpha_n/\xi_n\}$ where the ξ_i are the locations of the variables assigned by the statement. Empty statements have no effect on the state in which they are executed. Assignment statements result in a variant of the state in which they are executed by substituting the value of the right-hand side of the assignment for the location of the left-hand side of the assignment.

The **for** statement has probably the most interesting semantic definition. If the value of the upper bound is greater than or equal to the value of the lower bound in the state in which the **for** statement is being executed, then the statement in the loop will be executed at least once. The **for** statement then has the meaning of the same **for** statement, with one fewer iterations of the loop, followed by the statement in the **for** body, with the original value of the upper bound substituted for the loop

control variable. This has the effect of making any assignment to the loop control variable legal, but meaningless. For example, the statement `for` $i := 1$ `to` 10 `do` $x := 100; sum := sum + x$ `od` has the same meaning as the statement `for` $i := 1$ `to` 9 `do` $x := 100; sum := sum + x$ `od`; $x := 100; sum := sum + 10$. If the upper bound's value is less than the lower bound's value, then the `for` statement has the same meaning as the empty statement. Notice that since the loop bounds are fixed at the time of entrance to the loop, it is impossible to have infinite looping.

Definition 3.5.3 (Semantics of Stat)

$\mathcal{M}(v := s)(\sigma) = \sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v)(\sigma)\}$

$\mathcal{M}(S_1; S_2)(\sigma) = \mathcal{M}(S_2)\ (\mathcal{M}(S_1)(\sigma))$

$\mathcal{M}(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi})(\sigma) = \mathit{if}\ \mathcal{W}(b)(\sigma)\ \mathit{then}\ \mathcal{M}(S_1)(\sigma)\ \mathit{else}\ \mathcal{M}(S_2)(\sigma)\ \mathit{fi}$

$\mathcal{M}(D)(\sigma) = \sigma$

$\mathcal{M}(\texttt{for}\ x := s_1\ \texttt{to}\ s_2\ \texttt{do}\ S\ \texttt{od})(\sigma) =$

$\quad \mathit{if}\ \mathcal{R}(s_2)(\sigma) \geq \mathcal{R}(s_1)(\sigma)\ \mathit{then}$

$\quad\quad \mathcal{M}(\texttt{for}\ x := s_1\ \texttt{to}\ s_2\ \texttt{-}\ 1\ \texttt{do}\ S\ \texttt{od};\ S[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$

$\quad\quad \mathit{else}\ \mathcal{M}(D)(\sigma)\ \mathit{fi}$

Two statements are equal in a state if they have the same meaning in that state. They are *equal* if they have the same meaning in all states.

Definition 3.5.4 (Equality of statements)

Two statements S_1 *and* S_2 *are* equal in a state, *written* $S_1 =_\sigma S_2$, *if* $\mathcal{M}(S_1)(\sigma) = \mathcal{M}(S_2)(\sigma)$.

Two statements S_1 *and* S_2 *are* equal, *written* $S_1 = S_2$, *if for all states* $\sigma \in \Sigma$, $S_1 =_\sigma S_2$.

A transformation of a statement S_1 *into a statement* S_2 *is* valid *in state* σ *if* $S_1 =_\sigma S_2$.

A statement is said to be *nullable* if and only if it has no effect on any state. Nullable statements are of the form $x := x$, or some combination of nullable statements, such as `if` b `then` $x := x$ `else for` $y := s_1$ `to` s_2 `do` $y := y; z := z$ `od fi`.

Definition 3.5.5 (Nullable statements)

A statement S *is* nullable *iff* $S = D$.

3.6 Semantics of State Variants

Once the semantics of statements is established, some obvious results about state variants and their meanings can be proved. First, state variants can be interchanged when they refer to different locations.

Lemma 3.6.1 (Interchange of state variants)

$(\sigma\{\alpha_1/\xi_1\})\{\alpha_2/\xi_2\} = (\sigma\{\alpha_2/\xi_2\})\{\alpha_1/\xi_1\}$

Provided:

$\xi_1 \not\equiv \xi_2$

Proof:

It must be shown that $(\sigma\{\alpha_1/\xi_1\})\{\alpha_2/\xi_2\}(v) = (\sigma\{\alpha_2/\xi_2\})\{\alpha_1/\xi_1\}(v)\ \forall \xi \in \mathrm{LocV}$.

Case 1: $\xi \equiv \xi_1$

$(\sigma\{\alpha_1/\xi_1\})\{\alpha_2/\xi_2\}(\xi) = \sigma\{\alpha_1/\xi_1\}(\xi)$	(Def. of state variant[3.2.2])
$= \alpha_1$	(Def. of state variant[3.2.2])
$= \sigma\{\alpha_2/\xi_2\}(\alpha_1)$	(Def. of constants)
$= (\sigma\{\alpha_2/\xi_2\})\{\alpha_1/\xi_1\}(\xi)$	(Def. of state variant[3.2.2])

Case 2: $\xi \equiv \xi_2$

$(\sigma\{\alpha_1/\xi_1\})\{\alpha_2/\xi_2\}(\xi) = \sigma\{\alpha_1/\xi_1\}(\alpha_2)$	(Def. of state variant[3.2.2])
$= \alpha_2$	(Def. of constants)
$= \sigma\{\alpha_2/\xi_2\}(\xi)$	(Def. of state variant[3.2.2])
$= (\sigma\{\alpha_2/\xi_2\})\{\alpha_1/\xi_1\}(\xi)$	(Def. of state variant[3.2.2])

Case 3: $\xi \not\equiv \xi_1$ and $\xi \not\equiv \xi_2$

$(\sigma\{\alpha_1/\xi_1\})\{\alpha_2/\xi_2\}(\xi) = \sigma\{\alpha_1/\xi_1\}(\xi)$	(Def. of state variant[3.2.2])
$= \sigma(\xi)$	(Def. of state variant[3.2.2])
$= \sigma\{\alpha_2/\xi_2\}(\xi)$	(Def. of state variant[3.2.2])
$= (\sigma\{\alpha_2/\xi_2\})\{\alpha_1/\xi_1\}(\xi)$	(Def. of state variant[3.2.2])

□

The value of a variable in a state variant follows directly from the definition of state variants.

Lemma 3.6.2 (Semantics of state variants)

If the variable v is strictly non-self-referencing and v is not location-dependent on x then

$$\mathcal{R}(v)(\sigma\{\alpha/\mathcal{L}(x)(\sigma)\}) = \begin{cases} \alpha & \text{if } \mathcal{L}(v)\sigma = \mathcal{L}(x)(\sigma) \\ \mathcal{R}(v)(\sigma) & \text{otherwise} \end{cases}$$

Proof: This must be shown for all $v \in$ Ivar.

Case 1: $\mathcal{L}(v)(\sigma) = \mathcal{L}(x)(\sigma)$

$\mathcal{R}(v)(\sigma\{\alpha/\mathcal{L}(x)(\sigma)\})$	
$= \sigma\{\alpha/\mathcal{L}(x)(\sigma)\}(\mathcal{L}(v)(\sigma\{\alpha/\mathcal{L}(x)(\sigma)\}))$	(Def. of semantics of expressions[3.5.1])
$= \sigma\{\alpha/\mathcal{L}(x)(\sigma)\}(\mathcal{L}(v)(\sigma))$	(v is non-self-referencing)
$= \sigma\{\alpha/\mathcal{L}(x)(\sigma)\}(x)$	(Given)
$= \sigma\{\alpha/\mathcal{L}(x)(\sigma)\}(\mathcal{L}(x)(\sigma))$	(Def. of location of Svar [3.2.1])
$= \alpha$	(Def. of state variant[3.2.2])

Case 2: $\mathcal{L}(v)(\sigma) \neq \mathcal{L}(x)(\sigma)$

$\mathcal{R}(v)(\sigma\{\alpha/\mathcal{L}(x)(\sigma)\})$	
$= \sigma\{\alpha/\mathcal{L}(x)(\sigma)\}(\mathcal{L}(v)(\sigma\{\alpha/\mathcal{L}(x)(\sigma)\}))$	(Def. of semantics of expressions[3.5.1])
$= \sigma\{\alpha/\mathcal{L}(x)(\sigma)\}(\mathcal{L}(v)(\sigma))$	(Given)
$= \sigma(\mathcal{L}(v)(\sigma))$	(Def. of state variant[3.2.2])
$= \mathcal{R}(v)(\sigma)$	(Def. of semantics of expressions[3.5.1])

□

Substituting the value of a variable in a state, σ, for the location of that variable in σ does not change the meaning of σ.

Lemma 3.6.3

$\sigma\{\mathcal{R}(v)(\sigma)/\mathcal{L}(v)(\sigma)\} = \sigma$

Proof: It must be shown that both states have the same values for all $\xi \in$ LocV.

Case 1: $\xi = \mathcal{L}(v)(\sigma)$
$\sigma\{\mathcal{R}(v)(\sigma)/\mathcal{L}(v)(\sigma)\}(\xi) = \mathcal{R}(v)(\sigma)$ (Def. of state variant [3.2.2])
$= \sigma(\mathcal{L}(v)(\sigma))$ (Def. of semantics of expressions[3.5.1])
$= \sigma(\xi)$ (In this case, $\xi = \mathcal{L}(v)(\sigma)$)

Case 2: $\xi \neq \mathcal{L}(v)(\sigma)$
$\sigma\{\mathcal{R}(v)(\sigma)/\mathcal{L}(v)(\sigma)\}(\xi) = \sigma(\xi)$ (Def. of state variant [3.2.2])
□

A state variant may be added or removed if there is a subsequent variant referring to the same location.

Lemma 3.6.4 (Introduction and elimination of state variants)

$(\sigma\{\alpha_1/\xi\})\{\alpha_2/\xi\} = \sigma\{\alpha_2/\xi\}$

Proof: It must be shown that both states have the same values for all $\xi_1 \in$ LocV.

Case 1: $\xi_1 = \xi$
$(\sigma\{\alpha_1/\xi\})\{\alpha_2/\xi\}(\xi_1) = \alpha_2$ (Def. of state variant [3.2.2])
$= (\sigma\{\alpha_2/\xi\})(\xi_1)$ (Def. of state variant [3.2.2])

Case 2: $\xi_1 \neq \xi$
$(\sigma\{\alpha_1/\xi\})\{\alpha_2/\xi\}(\xi_1) = (\sigma\{\alpha_1/\xi\})(\xi_1)$ (Def. of state variant [3.2.2])
$= \sigma(\xi_1)$ (Def. of state variant [3.2.2])
$= (\sigma\{\alpha_2/\xi\})(\xi_1)$ (Def. of state variant [3.2.2])
□

3.7 Semantics of Substitution

Since substitution is used in defining the semantics of some statements, it is necessary to look at the resulting semantics of substitution. For boolean and integer

expressions, the value of the expression s (or b) after syntactic substitution for some variable x in a state is the same as the value of the expression in the state with the same semantic substitution made for the variable.

Lemma 3.7.1 (Semantics of substitution into expressions)

$\mathcal{R}(s[s_1/y])(\sigma) = \mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$

$\mathcal{W}(b[s_1/y])(\sigma) = \mathcal{W}(b)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$

Proof: By pairwise induction on the complexity of s and b. Since the operations **are assumed to preserve their meaning in any variant, only the basis steps are shown** here.

Case 1: $s \equiv y$

$\mathcal{R}(y[s_1/y])(\sigma) = \mathcal{R}(s_1)(\sigma)$	(Def. of substitution [3.4.1])
$= \mathcal{R}(y)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$	(Sem. of state variant [3.6.2])

Case 2: $s = x, x \not\equiv y$

$\mathcal{R}(x[s_1/y])(\sigma) = \mathcal{R}(x)(\sigma)$	(Def. of substitution [3.4.1])
$= \mathcal{R}(x)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$	(Sem. of state variant[3.6.2])

Case 3: $s = m$

$\mathcal{R}(m[s_1/y])(\sigma) = \alpha$	(Def. of constants [3.5.1])
$= \mathcal{R}(m)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$	(Def. of constants [3.5.1])

Case 4: $b = \texttt{true}$

$\mathcal{W}(\texttt{true}[s_1/y])(\sigma) = \mathbf{T}$	(Def. of `true` [3.5.2])
$= \mathcal{W}(\texttt{true})(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$	(Def. of `true` [3.5.2])

Case 5: $b = \texttt{false}$

$\mathcal{W}(\texttt{false}[s_1/y])(\sigma) = \mathbf{F}$	(Def. of `false` [3.5.2])
$= \mathcal{W}(\texttt{false})(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$	(Def. of `false` [3.5.2])

□

Substituting into locations is a bit more complicated. In this case, only variables can be substituted (since it makes no sense to discuss the location of an expression). Additionally, there are three possible cases. First, the original variable may be an indexed variable. (Recall that only simple, nonindexed variables can be substituted for, so no indexed variable will be replaced with another indexed variable.) Second, if the original variable is simple, there are two cases; the old location can be identical to it or always separate from it. Additionally, left-hand-side substitution has semantics very similar to full textual substitution into array variables.

Lemma 3.7.2 (Semantics of substitution into locations)

$$\mathcal{L}(v[v_1/y])(\sigma) = \begin{cases} \mathcal{L}(v_1)(\sigma) & \text{if } v \in \mathit{Svar},\ v \equiv y \\ \mathcal{L}(v)(\sigma) & \text{if } v \in \mathit{Svar},\ v \not\equiv y \\ \mathcal{L}(v)(\sigma\{\mathcal{R}(v_1)(\sigma)/\mathcal{L}(y)(\sigma)\}) & \text{if } v \in \mathit{Avar} \end{cases}$$

$$\mathcal{L}(v < v_1/y >)(\sigma) = \mathcal{L}(v)(\sigma\{\mathcal{R}(v_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$$

Proof: There are five possible cases (the first three for regular substitution and the last two for left-hand-side substitution).

Case 1: $v \equiv y$

$\mathcal{L}(y[v_1/y])(\sigma) = \mathcal{L}(y)(\sigma)$ (Def. of substitution [3.4.1])

Case 2: $v \in$ Svar, $v \not\equiv y$

$\mathcal{L}(v[v_1/y])(\sigma) = \mathcal{L}(v)(\sigma)$ (Def. of substitution [3.4.1])

Case 3: $v \in$ Avar, $v = a[s]$

$\mathcal{L}(a[s][v_1/y])(\sigma) = \mathcal{L}(a[s[v_1/y]])(\sigma)$ (Def. of substitution [3.4.1])

$= < a, \mathcal{R}(s[v_1/y])(\sigma) >$ (Def. of location of Avar [3.2.1])

$= < a, \mathcal{R}(s)(\sigma\{\mathcal{R}(v_1)(\sigma)/\mathcal{L}(y)(\sigma)\}) >$ (Sem. of subst. into expr. [3.7.1])

$= \mathcal{L}(a[s])(\sigma\{\mathcal{R}(v_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$ (Def. of location of Avar [3.2.1])

Case 4: $v \in \text{Svar}$

$\mathcal{L}(v <v_1/y>)(\sigma) = \mathcal{L}(v)(\sigma)$ (Def. of left-hand-side subst. [3.4.3])

$= \mathcal{L}(v)(\sigma\{\mathcal{R}(v_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$ (Def. of location of Svar [3.2.1])

Case 5: $v \in \text{Avar}$, $v = a[s]$

$\mathcal{L}(a[s] <v_1/y>)(\sigma) = \mathcal{L}(a[s[v_1/y]])(\sigma)$ (Def. of left-hand-side subst. [3.4.3])

$= \mathcal{L}(a[s])(\sigma\{\mathcal{R}(v_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$ (Same arguments as Case 3, above)

□

Defining substitution into statements also involves moving the substitution from the statement into the state. However, since the meaning of a statement results not in an integer or boolean value, but in another state, simply evaluating the meaning of the statement at a modified state, one with the new value substituted for the old, could possibly result in an incorrect new state, one which has the same modification. For example, the statement $(x := y)[3/y]$ evaluated in σ may seem to have the same results as evaluating $x := y$ in $\sigma\{3/y\}$. Both $\mathcal{M}((x := y)[3/y])(\sigma)$ and $\mathcal{M}(x := y)(\sigma\{3/y\})$ map x to 3. But $\mathcal{M}((x := y)[3/y])(\sigma)$ maps y to whatever value it has in σ, whereas $\mathcal{M}(x := y)(\sigma\{3/y\})$ maps y to 3. Thus, in the cases where the statement does not reset the old value, it is necessary to reset the old value after evaluating the statement in the state in which the substitution has taken place. This is described in the lemma below.

Lemma 3.7.3 (Semantics of substitution into statements)

$$\mathcal{M}(S[s_1/y])(\sigma) = \begin{cases} (\mathcal{M}(S)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\}))\{\mathcal{R}(y)(\sigma)/\mathcal{L}(y)(\sigma)\} \\ \quad \textit{if } \mathcal{R}(y)(\sigma) = \mathcal{R}(y)(\mathcal{M}(S)(\sigma)) \\ \mathcal{M}(S)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\}) \\ \quad \textit{otherwise} \end{cases}$$

Proof: By induction on the complexity of S. Only the most interesting case, when S is an assignment statement, is presented.

Case 1: $\mathcal{R}(y)(\sigma) \neq \mathcal{R}(y)(\mathcal{M}(S)(\sigma))$.
(Since S is an assignment statement, it must be of the form $y := s$.)
$\mathcal{M}((y := s)[s_1/y])(\sigma)$
$= \mathcal{M}(y <s_1/y>:= s[s_1/y])(\sigma)$ (Def. of substitution [3.4.4])
$= \mathcal{M}(y := s[s_1/y])(\sigma)$ (Def. of l.h.s. subst. [3.4.3])
$= \sigma\{\mathcal{R}(s[s_1/y])(\sigma)/\mathcal{L}(y)(\sigma)\}$ (Def. of $v := s$ [3.5.3])
$= \sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(y)(\sigma)\}$
(Sem. of substitution [3.7.1])
$= (\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(y)(\sigma)\}$
(Introduction of a variant [3.6.4])
$= (\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$
$\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(y)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})\}$
(Def. of location of Svar [3.2.1])
$= \mathcal{M}(y := s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$ (Def. of $v := s$ [3.5.3])

Case 2: $\mathcal{R}(y)(\sigma) = \mathcal{R}(y)(\mathcal{M}(S)(\sigma))$. (Thus S is of the form $v := s$ and $y \not\equiv v$.)
$\mathcal{M}((v := s)[s_1/y])(\sigma)$
$= \mathcal{M}(v <s_1/y>:= s[s_1/y])(\sigma)$ (Def. of substitution [3.4.4])
$= \sigma\{\mathcal{R}(s[s_1/y])(\sigma)/\mathcal{L}(v <s_1/y>)(\sigma)\}$ (Def. of $v := s$ [3.5.3])
$= \sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(v <s_1/y>)(\sigma)\}$
(Sem. of substitution [3.7.1])
$= \sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(v)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})\}$
(Sem. of substitution [3.7.2])
$= (\sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(v)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})\})$
$\{\mathcal{R}(y)(\sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(v)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})\})/$
$\mathcal{L}(y)(\sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(v)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})\})\}$
(Sem. of state variants [3.6.2])
$= (\sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(v)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})\})$
$\{\mathcal{R}(y)(\sigma)/$
$\mathcal{L}(y)(\sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(v)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})\})\}$
(Sem. of state variants [3.6.2])
$= (\sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(v)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})\})$
$\{\mathcal{R}(y)(\sigma)/\mathcal{L}(y)(\sigma)\}$ (Def. of location of Svar [3.2.1])
$= ((\sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(v)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})\})$
$\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\}\}\{\mathcal{R}(y)(\sigma)/\mathcal{L}(y)(\sigma)\}$
(Introduction of a variant [3.6.4])
$= ((\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})$
$\{\mathcal{R}(s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})/\mathcal{L}(v)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\})\})$
$\{\mathcal{R}(y)(\sigma)/\mathcal{L}(y)(\sigma)\}$ (Interchange of state variants [3.6.1])
$= (\mathcal{M}(v := s)(\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(y)(\sigma)\}))\{\mathcal{R}(y)(\sigma)/\mathcal{L}(y)(\sigma)\}$
(Def. of $v := s$ [3.5.3])

□

Although substitution into assignment statements uses a special left-hand-side substitution, in some cases (most particularly, copy propagation), full textual variable substitution, rather than left-hand-side substitution, occurs in the left-hand-side of assignment statements. This strong substitution will result in one of two cases, depending on whether or not the original variable is identical to the old variable.

Lemma 3.7.4 (Semantics of strong substitution into statements)

If v_2 is location-independent of v_1 then

$$\mathcal{M}(v_1[v_2/x] := s[v_2/x])(\sigma) = \begin{cases} \mathcal{M}(v_2 := s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\}) & \text{if } v_1 \in Svar \\ & \text{and } v_1 \equiv x \\ (\mathcal{M}(v_1 := s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})) & \\ \quad \{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\} & \text{otherwise} \end{cases}$$

Proof: There are three possible cases for $\mathcal{L}(v_1)$ and $\mathcal{L}(x)$.

Case 1: $v_1 \in$ Svar, $v_1 \equiv x$.

$\mathcal{M}(v_1[v_2/x] := s[v_2/x])(\sigma)$

$= \sigma\{\mathcal{R}(s[v_2/x])(\sigma)/\mathcal{L}(v_1[v_2/x])(\sigma)\}$ (Def. of $v := s$ [3.5.3])

$= \sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_1[v_2/x])(\sigma)\}$
(Sem. of substitution [3.7.1])

$= \sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_2)(\sigma)\}$
(Sem. of substitution [3.7.2])

$= \sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_2)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\}$
(Given)

$= \sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_2)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})\}$
(Given)

$= \mathcal{M}(v_2 := s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})$ (Def. of $v := s$ [3.5.3])

Case 2: $v_1 \in$ Svar, $v_1 \not\equiv x$.

$\mathcal{M}(v_1[v_2/x] := s[v_2/x])(\sigma)$

$= \mathcal{M}(v_1 := s[v_2/x])(\sigma)$ (Def. of substitution [3.4.1])

$= \sigma\{\mathcal{R}(s[v_2/x])(\sigma)/\mathcal{L}(v_1)(\sigma)\}$ (Def. of $v := s$ [3.5.3])

$= \sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_1)(\sigma)\}$
(Sem. of substitution [3.7.1])

$= (\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})\{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_1)(\sigma)\}$
(Introduction of a variant [3.6.4])

$$
\begin{aligned}
&= ((\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\}\\
&\quad \{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_1)(\sigma)\}\\
&\qquad\qquad \text{(Sem. of state variants [3.6.3])}\\
&= ((\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})\{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_1)(\sigma)\})\\
&\quad \{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\} \qquad \text{(Interchange of state variants [3.6.1])}\\
&= ((\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})\\
&\quad \{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_1)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})\})\\
&\qquad \{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\} \qquad \text{(Given)}\\
&= (\mathcal{M}(v_1 := s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\}\\
&\qquad\qquad \text{(Def. of } v := s \text{ [3.5.3])}
\end{aligned}
$$

Case 3: $v_1 \in \text{Avar}$.

$$
\begin{aligned}
&\mathcal{M}(v_1[v_2/x] := s[v_2/x])(\sigma)\\
&\quad = \sigma\{\mathcal{R}(s[v_2/x])(\sigma)/\mathcal{L}(v_1[v_2/x])(\sigma)\} \qquad \text{(Def. of } v := s \text{ [3.5.3])}\\
&\quad = \sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_1[v_2/x])(\sigma)\}\\
&\qquad\qquad \text{(Sem. of substitution [3.7.1])}\\
&\quad = \sigma\{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_1)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})\}\\
&\qquad\qquad \text{(Sem. of substitution [3.7.2])}\\
&\quad = (\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})\\
&\qquad \{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_1)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})\}\\
&\qquad\qquad \text{(Sem. of state variants [3.6.3])}\\
&\quad = ((\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})\\
&\qquad \{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_1)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})\}\\
&\qquad\qquad \text{(Introduction of a variant [3.6.4])}\\
&\quad = ((\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})\\
&\qquad \{\mathcal{R}(s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(v_1)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\})\})\\
&\qquad\quad \{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\} \qquad \text{(Interchange of state variants [3.6.1])}\\
&\quad = (\mathcal{M}(v_1 := s)(\sigma\{\mathcal{R}(v_2)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\}\\
&\qquad\qquad \text{(Def. of } v := s \text{ [3.5.3])}
\end{aligned}
$$

□

3.8 *Sets* and *Uses*

In examining statements for possible transformations, it is often necessary to see which variables they manipulate. Statements can manipulate variables in two different ways—they may simply reference the variable's value without changing it, or they may actually give a variable a new value. The first will be referred to as a

use of the variable and the second a *set* of the variable. A definition of *using* and *setting* variables is given below:

Definition 3.8.1 (Setting and using)

Let $\Phi \in \Sigma \rightarrow \Sigma$ *and* $x \in Ivar$.

a) Φ sets x *whenever, for some* σ, $\Phi(\sigma)(x) \neq (\sigma)(x)$.

b) Φ uses x *whenever, for some* σ *and* α, $\Phi(\sigma\{\alpha/\mathcal{L}(x)(\sigma)\}) \neq \Phi(\sigma)\{\alpha/\mathcal{L}(x)(\sigma)\}$.

The set of all variables set *and* used *by a given function* Φ, *will be written* sets(Φ) *and* uses(Φ) *respectively.*

While expressions do not set variables' values, they certainly do use the values of variables, so the definition of *uses* is extended to the functions $\mathcal{R}$ and $\mathcal{W}$ in the following definition.

Definition 3.8.2 (Using)

$\mathcal{R}(s)$ uses x *whenever, for some* α *and* σ, $\mathcal{R}(s)(\sigma\{\alpha/\mathcal{L}(x)(\sigma)\}) \neq \mathcal{R}(s)(\sigma)$. *The set of all variables* used *by* $\mathcal{R}(s)$ *will be written* uses *(s).*

$\mathcal{W}(b)(\sigma)$ uses x *whenever, for some* α *and* σ, $\mathcal{W}(b)(\sigma\{\alpha/\mathcal{L}(x)(\sigma)\}) \neq \mathcal{W}(b)(\sigma)$. *The set of all variables* used *by* $\mathcal{W}(b)$ *will be written* uses *(b).*

If an assignment statement $v := s$ is not nullable, then *uses* (v) and *uses* (s) are subsets of *uses* $(v := s)$.

The integer and boolean operations in this language have the same meanings in all states, so if a variant does not change the locations of the variables or values at those locations used by an expression, it will not change the value of the expression.

Lemma 3.8.1 (Equality under state variants)

If $x \notin$ uses(s) *then* $\mathcal{R}(s)(\sigma\{\alpha/x\}) = \mathcal{R}(s)(\sigma)$.

If $x \notin$ uses(b) *then* $\mathcal{W}(b)(\sigma\{\alpha/x\}) = \mathcal{W}(b)(\sigma)$.

Proof: By induction on the complexity of s and b. Since the operations are assumed to preserve their meaning in any variant, only the basis steps are shown here.

Case 1: $s = y, y \neq x$

$\mathcal{R}(y)(\sigma\{\alpha/x\}) = \mathcal{R}(y)(\sigma)$ (Sem. of state variant[3.6.2])

Case 2: $s = m$

$\mathcal{R}(m)(\sigma\{\alpha/x\}) = \alpha_1$ (Def. of constants [3.5.1], where m = α_1)

$= \mathcal{R}(m)(\sigma)$ (Def. of constants [3.5.1])

Case 3: $b =$ `true`

$\mathcal{W}(\texttt{true})(\sigma\{\alpha/x\}) = \mathbf{T}$ (Def. of `true` [3.5.2])

$= \mathcal{W}(\texttt{true})(\sigma)$ (Def. of `true` [3.5.2])

Case 4: $b =$ `false`

$\mathcal{W}(\texttt{false})(\sigma\{\alpha/x\}) = \mathbf{F}$ (Def. of `false` [3.5.2])

$= \mathcal{W}(\texttt{false})(\sigma)$ (Def. of `false` [3.5.2])

□

When actually performing transformations on code, *sets* and *uses* (or more likely, their static equivalents, discussed in Section 3.9) will be computed to determine whether a particular transformation is valid. Knowing that the intersection of the set of all variables set by one statement with the set of all variables used by another statement is empty provides additional information about the statements, as described in the lemmas below. If the *sets* of an arbitrary statement and the *uses* of an assignment statement have an empty intersection, then the value and location of the left-hand side of the assignment and the value of the right-hand side of the assignment will not change as a result of executing the statement.

Lemma 3.8.2 (Empty intersections of sets *and* uses*)*

If sets$(\mathcal{M}(S)) \cap$ uses$(\mathcal{M}(v := s)) = \phi$ *and* $v := s$ *is not nullable, then*

- $\mathcal{L}(v)(\sigma) = \mathcal{L}(v)(\mathcal{M}(S)(\sigma))$
- $\mathcal{R}(v)(\sigma) = \mathcal{R}(v)(\mathcal{M}(S)(\sigma))$
- $\mathcal{R}(s)(\sigma) = \mathcal{R}(s)(\mathcal{M}(S)(\sigma))$

For the proof of this, assume $\mathcal{M}(S)(\sigma) = \sigma\{\alpha_1/\xi_1\}\ldots\{\alpha_n/\xi_n\}$ where $\sigma(\xi_i) \neq \alpha_i$ (so that the ξ_i are only the values set by $\mathcal{M}(S)$).

Proof of the first part:

Case 1: $v = x \in$ Svar

$\mathcal{L}(x)(\sigma) = \mathcal{L}(x)(\mathcal{M}(S)(\sigma))$ (Def. of location of Svar [3.2.1])

Case 2: $v = a[s] \in$ Avar

$\mathcal{L}(a[s])(\sigma) =< a, \mathcal{R}(s)(\sigma) >$ (Def. of location of Avar [3.2.1])

$= < a, \mathcal{R}(s)(\sigma\{\alpha_1/\xi_1\}) >$ (Def. of *sets* and *uses* [3.8.1])

$= < a, \mathcal{R}(s)(\sigma\{\alpha_1/\xi_1\}\ldots\{\alpha_n/\xi_n\}) >$ (Repeated def. of *sets* and *uses* [3.8.1])

$= < a, \mathcal{R}(s)(\mathcal{M}(S)(\sigma)) >$ (Given)

$= \mathcal{L}(a[s])(\mathcal{M}(S)(\sigma))$ (Def. of location of Avar [3.2.1])

Proof of the second part:

$\mathcal{R}(v)(\sigma) = \mathcal{R}(v)(\sigma\{\alpha_1/\xi_1\})$ (Def. of *sets* and *uses* [3.8.1])

$= \mathcal{R}(v)(\sigma\{\alpha_1/\xi_1\}\ldots\{\alpha_n/\xi_n\})$ (Repeated def. of *sets* and *uses* [3.8.1])

$= \mathcal{R}(v)(\mathcal{M}(S)(\sigma))$ (Given)

The proof of the third part is almost identical to this.
□

Notice that the previous lemma requires that the statements in question not be nullable. If they were, this result may not apply. Consider the statement $x := x$. Since both *uses* and *sets* of $\mathcal{M}(x := x)$ are empty, the intersection of them with *uses* or *sets* of any other statement is also empty. Therefore, the intersection of *uses* ($\mathcal{M}(x := x)$) and *sets* ($\mathcal{M}(x := 7)$) is empty, but it is certainly not the case that $\mathcal{R}(x)(\sigma) = \mathcal{R}(x)(\mathcal{M}(x : -7)(\sigma))$. Counter-examples to the other two results are also readily available.

This result can be applied to the intersection of the set *sets* and the set *uses* of an arbitrary expression.

Lemma 3.8.3 (Empty intersections of sets *and* uses*)*

If sets$(\mathcal{M}(S))$ ∩ uses $(s) = \phi$ *then* $\mathcal{R}(s)(\sigma) = \mathcal{R}(s)(\mathcal{M}(S)(\sigma))$.

If sets$(\mathcal{M}(S))$ ∩ uses $(b) = \phi$ *then* $\mathcal{W}(b)(\sigma) = \mathcal{W}(b)(\mathcal{M}(S)(\sigma))$.

The proof of this lemma is similar to that of Lemma 3.8.2.

If the *sets* of an assignment statement and the *uses* of an arbitrary statement have an empty intersection, the meaning of the arbitrary statement is the same, whether or not the assignment statement has been executed. As with all cases in the definition of the meaning of a statement in a modified state, the modification must be undone after evaluating the statement.

Lemma 3.8.4 (Empty intersections of sets *and* uses*)*
If sets$(\mathcal{M}(v := s)) \cap$ uses$(\mathcal{M}(S)) = \phi$ *and* $v := s$ *is not nullable, then*
$\mathcal{M}(S)(\sigma) = (\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v)(\sigma)\})\{\mathcal{R}(v)(\sigma)/\mathcal{L}(v)(\sigma)\}$

This proof follows by mathematical induction on the complexity of S. The basis case when S is D follows directly from the meaning of D (in Definition 3.5.3). When S is an assignment statement, the basis case comes from Lemma 3.8.2.

In the most general case, if the intersection of the *sets* of an arbitrary statement, S_1, and the *uses* of another arbitrary statement, S_2, is empty, then the locations assigned by S_2 and the values assigned to those locations are the same in the state σ and $\mathcal{M}(S_1)(\sigma)$. This is clearly not reciprocal. If the intersection of *sets*(S_1) and *uses* (S_2) is empty, then it is possible that the locations or the values assigned by S_1 may be different in σ and $\mathcal{M}(S_2)(\sigma)$. Consider if S_1 were $x := y$ and S_2 were $y := 15$. Here, *sets*(S_1) is $[x]$ and *uses*(S_2) is $[y]$. The effective meaning of S_2 in any state, adding a variant with 15 in place of y, is the same regardless of whether S_1 has been executed. However, if S_2 is not executed before S_1, S_1 yields a variant where $\sigma(y)$ is

assigned to x, whereas, if S_2 is executed first, 15 is assigned to y. This is expressed formally in the following lemma.

Lemma 3.8.5 (Empty intersections of sets *and* uses*)*

If $\text{sets}(\mathcal{M}(S_1)) \cap \text{uses}(\mathcal{M}(S_2)) = \phi$ *then given* σ *where* $\mathcal{M}(S_1)(\sigma) = \sigma\{\alpha_{11}/\xi_{11}\}\ldots\{\alpha_{1m}/\xi_{1m}\}$ *and* $\mathcal{M}(S_2)(\sigma) = \sigma\{\alpha_{21}/\xi_{21}\}\ldots\{\alpha_{2n}/\xi_{2n}\}$ *and* $\sigma(\xi_{ij}) \neq \alpha_{ij}$*, then*

- $\forall i \forall j (\xi_{1i} \neq \xi_{2j})$
- $\mathcal{M}(S_2)(\mathcal{M}(S_1)(\sigma)) = (\sigma\{\alpha_{21}/\xi_{21}\}\ldots\{\alpha_{2n}/\xi_{2n}\})\{\alpha_{11}/\xi_{11}\}\ldots\{\alpha_{1m}/\xi_{1m}\}$

Proving that the ξ_{ij} are not equal uses a simple contradiction from the definitions of *sets*, *uses*, and the meaning of a statement. The proof of the second part follows by mathematical induction on the complexity of S_2. The basis case when S is D follows directly from the meaning of D (in Definition 3.5.3). When S is an assignment statement, the basis case comes from Lemma 3.8.2.

Notice that since the ξ_{1i} and the ξ_{2j} are not equal, Lemma 3.6.1 can be used to interchange the state variants, giving, for example

$$\mathcal{M}(S_2)(\mathcal{M}(S_1)(\sigma)) = (\sigma\{\alpha_{11}/\xi_{11}\}\ldots\{\alpha_{1m}/\xi_{1m}\})\{\alpha_{21}/\xi_{21}\}\ldots\{\alpha_{2n}/\xi_{2n}\}.$$

3.9 Static Approximation of *Sets* and *Uses*

The definitions of *sets* and *uses* only consider dynamic cases where the state's value is known in advance. To actually perform transformations, however, it is necessary to evaluate the *sets* and *uses* of a statement or expression statically with little, if any, information about the state in which the statement or expression is being evaluated. The function *ivar* is used as a static approximation of *sets*.

Definition 3.9.1 (ivar)

The set of all integer variables occurring in s, b, or S is denoted by ivar(s), ivar(b) *or* ivar(S), *respectively, and consists of the following:*

$ivar(x) \equiv \{x\}$,

$ivar(a[s]) \equiv \{a\} \cup ivar(s)$,

$ivar(m) \equiv \phi$

$ivar(s_1 \oplus s_2) \equiv ivar(s_1) \cup ivar(s_2)$,

$ivar(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}) \equiv ivar(b) \cup ivar(s_1) \cup ivar(s_2)$,

$ivar(\texttt{true}) \equiv \phi$

$ivar(\texttt{false}) \equiv \phi$

$ivar(s_1 \ominus s_2) \equiv ivar(s_1) \cup ivar(s_2)$,

$ivar(\neg b) \equiv ivar(b)$

$ivar(v := s) \equiv ivar(v) \cup ivar(s)$,

$ivar(S_1; S_2) \equiv ivar(S_1) \cup ivar(S_2)$,

$ivar(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) \equiv ivar(B) \cup ivar(S_1) \cup ivar(S_2)$,

$ivar(\mathtt{D}) \equiv \phi$, *and*

$ivar(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ do } S \texttt{ od}) \equiv ivar(s_1) \cup ivar(s_2) \cup ivar(S)$.

Lemma 3.9.1

if $x \notin ivar(S)$ *then* $x \notin \text{uses}(\mathcal{M}(S))$

This proof follows by induction on complexity of S.

The function *livar*, defined only for statements, describes the set of variables on the left-hand side of assignments and can be used statically in place of *uses*.

Definition 3.9.2 (livar)

The set of all integer variables on the left-hand-side of an assignment in S is denoted by livar(S) and consists of the following:

$livar(x := s) \equiv \{x\}$,

$livar(a[s] := s) \equiv \{a\}$,

$livar(S_1; S_2) \equiv livar(S_1) \cup livar(S_2)$,

$livar(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) \equiv livar(S_1) \cup livar(S_2)$,

$livar(\texttt{D}) \equiv \phi$, *and*

$livar(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ do } S \texttt{ od}) \equiv livar(S)$.

Lemma 3.9.2

if $x \notin livar(S)$ *then* $x \notin \text{sets}(\mathcal{M}(S))$

This proof follows by induction on complexity of S.

While these definitions provide a close approximation of the simple variables in *sets* and *uses*, they tend to be overly broad with array references, adding the entire array to the set, instead of only the element being accessed. For example, ivar $(x := 3 * w + z)$ is $[x, w, z]$, but ivar $(a[x] := 3 * w + z)$ is $[a, x, w, z]$. Since array references are so important and prevalent in image processing, a finer approximation of *sets* and *uses* is employed in the actual implementation of these transformations. This approximation is discussed in detail in Section 6.3.

For simple variables, the sets $uses(\mathcal{M}(S))$ and $ivar(S)$ (as well as the sets $livar(S)$ and $sets(\mathcal{M}(S))$) may appear to be equivalent, and very often do refer to the same set. They are not, however, always identical. As a counter-example, consider the

statement $y := x - x$. Here, the set $ivar(S)$ is $[x, y]$, those variables which appear in the statement. But the set $uses(\mathcal{M}(S))$ is $[y]$. It does not include x because $\mathcal{M}(S)(\sigma\{\alpha/x\}) = (\mathcal{M}(S)(\sigma))\{\alpha/x\}$ for all $\sigma \in \Sigma$ and all $\alpha \in V$. Using *ivar* as an approximation of *uses* (and *livar* as an approximation of *sets*) will often cause no problem, but it may prevent some transformations from taking place. For instance, if $ivar(S)$ is employed instead of $uses(\mathcal{M}(S))$ in the cases for statement interchange (in Theorem 4.1.1), the statements $x := 7; y := x - x$ are not interchangeable.

The sets *ivar* and *livar* usually give a reasonable approximation of always separate.

Lemma 3.9.3 (Static approximation of Always Separate)

If $ivar(S_1)$ *and* $livar(S_2)$ *are disjoint, the elements of* uses$(\mathcal{M}(S_1))$ *and* sets$(\mathcal{M}(S_2))$ *are always separate.*

Proof:

Let $ivar(S_1) \cap livar(S_2) = \phi$ and $v \in uses(\mathcal{M}(S_1))$ and $sets\ (\mathcal{M}(S_2))$
By lemma 3.9.2, since $v \in uses(\mathcal{M}(S_1))$, $v \in ivar(S_1)$
By lemma 3.9.1, since $v \in sets(\mathcal{M}(S_2))$, $v \in livar(S_2)$
Therefore, $v \in ivar(S_1) \cap livar(S_2)$.
By contradiction, there can be no v such that $v \in uses(\mathcal{M}(S_1))$ and $sets(\mathcal{M}(S_2))$, so they are always separate.

□

CHAPTER 4
PRIMITIVE TRANSFORMATIONS

This chapter focuses on the minimal transformations that can be performed on programs in the language defined in Chapter 3. These transformations will be combined in Chapter 5 to give global optimizations. Since they are less complex than the global optimizations, primitive transformations are more easily proved correct. The proofs of the correctness of each of the minimal transformations is provided here as well. The basic code transformations are:

- statement interchange (Theorem 4.1.1).

  There are two variations on interchange:

  - interchange with substitution (Theorem 4.1.4)
  - interchange with backward substitution (Theorem 4.1.5)

- statement compression (Theorem 4.1.6)
- movement of statements into (and out of) `if` statements (Theorem 4.2.1) and a variation that uses substitution (Theorem 4.2.2)
- **`if then else`** statement splitting (Theorem 4.2.3)
- **`if then else`** statement simplification (Theorem 4.2.5)
- loop rolling and unrolling (Theorems 4.3.1 and 4.3.2)

To move the statement S_k in front of the statement S_1,
it must be interchanged with all of the statements $S_1, \ldots, S_{k-1}$:

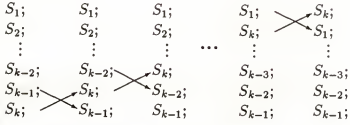

Figure 4.1. Statement Interchange Used to Move a Statement

4.1 Statement Transformations

The most basic of the smaller transformations is statement interchange. Two adjacent statements may, in some situations, be interchanged with no ultimate change in meaning to the entire program. In some computer architectures this may result in a shorter running time, but that is not the goal of statement interchange. Instead, it will support rearranging code so that other, more powerful optimizations may be performed. Statement interchange is often used to move one statement to the beginning (or end) of a group of other statements, as shown in figure 4.1. In code motion, a standard loop optimization technique that removes invariant statements from loops, statement interchanges first move the statement being removed from the loop to the beginning of that loop. Additionally, many of the other transformations discussed in this section require that statements be interchangeable in order for the transformation to take place.

Definition 4.1.1 (Interchangeability)

Two statements S_1 and S_2 are said to be interchangeable *in state σ if $\mathcal{M}(S_1; S_2)(\sigma)$ = $\mathcal{M}(S_2; S_1)(\sigma)$.*

While there will always be pathological cases in which two seemingly conflicting statements will still be interchangeable, a set of sufficient conditions for statement interchange are given in theorem 4.1.1.

Theorem 4.1.1 (Conditions for statement interchange)

Two statements S_1 and S_2 are interchangeable if any of the following conditions are true:

a) $S_1 = S_2$

b) $\forall y \forall x \ni y \in \text{sets}(\mathcal{M}(S_1))$, *and* $x \in \text{uses}(\mathcal{M}(S_2))$, *sep(y, x);*

$\forall y \forall x \ni y \in \text{sets}(\mathcal{M}(S_2))$, *and* $x \in \text{uses}(\mathcal{M}(S_1))$, *sep(y, x);*

c) $S_1 = v := f(v)$, $S_2 = v := g(v)$, *v is strictly non-self-referencing, and* $f(g(v)) = g(f(v))$.

Proof:

The proof of part a) follows directly from the definition of equal statements [3.5.4].

The proof of part b) is in Theorem 4.1.2.

The proof of part c) is in Theorem 4.1.3.

Clearly, if two statements are identical, their order does not matter. It is when the statements are different that statement interchange becomes interesting. First, two statements can be interchanged if the locations set by each are not used by the other. For example, $a[x] := 3 * y$ can be interchanged with $z := w - y$, because a, which is set by the first, is not used in computing $w - y$; and z, which is set by the second, is not used in computing $3 * y$.

Theorem 4.1.2 (Statement interchange)

$\models S_1; S_2 \;=\; S_2; S_1$

Provided:

$\text{sets}(\mathcal{M}(S_1)) \cap \text{uses}(\mathcal{M}(S_2)) = \phi$

$\text{sets}(\mathcal{M}(S_2)) \cap \text{uses}(\mathcal{M}(S_1)) = \phi$

Proof:
Assume that $\mathcal{M}(S_1)(\sigma) = \sigma\{\alpha_{11}/\xi_{11}\}\ldots\sigma\{\alpha_{1m}/\xi_{1m}\}$ and
that $\mathcal{M}(S_2)(\sigma) = \sigma\{\alpha_{21}/\xi_{21}\}\ldots\sigma\{\alpha_{2n}/\xi_{2n}\}$ where $\sigma(\xi_{xy}) \neq \alpha_{xy}$.
$\mathcal{M}(S_1; S_2)(\sigma)$

$= \mathcal{M}(S_2)(\mathcal{M}(S_1)(\sigma))$ (Def. of $v := s$ [3.5.3])

$= (\sigma\{\alpha_{11}/\xi_{11}\}\ldots\sigma\{\alpha_{1m}/\xi_{1m}\})\{\alpha_{21}/\xi_{21}\}\ldots\sigma\{\alpha_{2n}/\xi_{2n}\}$
(Def. of empty *sets* and *uses* [3.8.5])

$= (\sigma\{\alpha_{21}/\xi_{21}\}\ldots\sigma\{\alpha_{2n}/\xi_{2n}\})\{\alpha_{11}/\xi_{11}\}\ldots\sigma\{\alpha_{1m}/\xi_{1m}\}$
(Interchange of state variants [3.6.1])

$= \mathcal{M}(S_1)(\mathcal{M}(S_2)(\sigma))$ (Def. of empty *sets* and *uses* [3.8.5])

$= \mathcal{M}(S_2; S_1)(\sigma)$ (Def. of $v := s$ [3.5.3])

□

It is not necessary that two statements have empty intersections of their *sets* and *uses* sets in order to interchange them. If the statements are assignments to the same variable and the expressions being assigned to the variable can be composed in either order (i.e. if $f(g(v)) = g(f(v))$), the statements may also be interchanged. For example, the statements $x := 3 * x$ and $x := 4 * x$ may be interchanged because, for integer expressions, $4 * (3 * x) = 3 * (4 * x)$. These statements would not be interchangeable under the conditions of Theorem 4.1.2 because the *uses* and the *sets* of each statement is identical, the set containing just x. The statements $x := 3 * x$ and $x := 4 + x$ are obviously not interchangeable, and would not be as a result of this theorem, because, in general, $4 + (3 * x) \neq 3 * (4 + x)$.

Theorem 4.1.3 (Interchange of assignments of functions)

$\models v := f(v); v := g(v) \quad = \quad v := g(v); v := f(v)$

Provided:

v is strictly non-self-referencing and $f(g(v)) = g(f(v))$

Proof: This can be proved by manipulating the meaning of v := f(v); v := g(v).

$\mathcal{M}(v := f(v); v := g(v))(\sigma)$
$= \mathcal{M}(v := g(v))(\mathcal{M}(v := f(v))(\sigma))$ (Def. of $S_1; S_2$ [3.5.3])
$= \mathcal{M}(v := g(v))(\sigma\{\mathcal{R}(f(v))(\sigma)/\mathcal{L}(v)(\sigma)\})$
(Def. of $v := s$ [3.5.3])
$= (\sigma\{\mathcal{R}(f(v))(\sigma)/\mathcal{L}(v)(\sigma)\})$
$\{\mathcal{R}(g(v))(\sigma\{\mathcal{R}(f(v))(\sigma)/\mathcal{L}(v)(\sigma)\})/\mathcal{L}(v)(\sigma\{\mathcal{R}(f(v))(\sigma)/\mathcal{L}(v)(\sigma)\})\}$
(Def. of $v := s$ [3.5.3])
$= (\sigma\{\mathcal{R}(f(v))(\sigma)/\mathcal{L}(v)(\sigma)\})$
$\{\mathcal{R}(g(v)[f(v)/v])(\sigma)/\mathcal{L}(v)(\sigma\{\mathcal{R}(f(v))(\sigma)/\mathcal{L}(v)(\sigma)\})\}$
(Sem. of subst. into expr. [Lemma 3.7.1])
$= (\sigma\{\mathcal{R}(f(v))(\sigma)/\mathcal{L}(v)(\sigma)\})\{\mathcal{R}(g(v)[f(v)/v])(\sigma)/\mathcal{L}(v)(\sigma)\}$
(Given)
$= \sigma\{\mathcal{R}(g(v)[f(v)/v])(\sigma)/\mathcal{L}(v)(\sigma)\}$ (Elimination of a variant [3.6.4])
$= \sigma\{\mathcal{R}(g(f(v)))(\sigma)/\mathcal{L}(v)(\sigma)\}$ (Def. of substitution [3.4.1])
$= \sigma\{\mathcal{R}(f(g(v)))(\sigma)/\mathcal{L}(v)(\sigma)\}$ (Given)
$= \sigma\{\mathcal{R}(f(v)[g(v)/v])(\sigma)/\mathcal{L}(v)(\sigma)\}$ (Def. of substitution [3.4.1])
$= (\sigma\{\mathcal{R}(g(v))(\sigma)/\mathcal{L}(v)(\sigma)\})\{\mathcal{R}(f(v)[g(v)/v])(\sigma)/\mathcal{L}(v)(\sigma)\}$
(Introduction of a variant [3.6.4])
$= (\sigma\{\mathcal{R}(g(v))(\sigma)/\mathcal{L}(v)(\sigma)\})\{\mathcal{R}(f(v))(\sigma\{\mathcal{R}(g(v))(\sigma)/\mathcal{L}(v)(\sigma)\})/\mathcal{L}(v)(\sigma)\}$
(Sem. of subst. into expr. [Lemma 3.7.1])
$= (\sigma\{\mathcal{R}(g(v))(\sigma)/\mathcal{L}(v)(\sigma)\})$
$\{\mathcal{R}(f(v))(\sigma\{\mathcal{R}(g(v))(\sigma)/\mathcal{L}(v)(\sigma)\})/\mathcal{L}(v)(\sigma\{\mathcal{R}(g(v))(\sigma)/\mathcal{L}(v)(\sigma)\})\}$
(Given)
$= (\mathcal{M}(v := f(v))(\sigma\{\mathcal{R}(g(v))(\sigma)/\mathcal{L}(v)(\sigma)\})$
(Def. of $v := s$ [3.5.3])
$= (\mathcal{M}(v := f(v))(\mathcal{M}(v := g(v))(\sigma))$ (Def. of $v := s$ [3.5.3])
$= (\mathcal{M}(v := g(v); v := f(v))(\sigma))$ (Def. of $S_1; S_2$ [3.5.3])

□

Sometimes the conditions necessary for interchange do not exist. If the first of the pair of statements is an assignment statement, it may still be possible and desirable to rearrange the code by interchanging the statements while making the

same substitution that would have been made by the assignment statement. This interchange with substitution is discussed in the following theorem.

Theorem 4.1.4 (Interchange with substitution)

$\models x := s; S \quad = \quad S[s/x]; x := s$

Provided:

$\text{uses}(\mathcal{M}(x := s)) \cap \text{sets}(\mathcal{M}(S)) = \phi$

Proof: This can be proved by manipulation of the meaning of $x := s; S$. If $x := s$ is nullable, the result is obviously true ($x := x; S = S[x/x]; x := x$), so the proof assumes $x := s$ is not nullable.

As a preliminary result, notice that $\mathcal{R}(s)((\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\}) = \mathcal{R}(s)((\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})$. This is not the result which would be expected as a result of Lemma 3.8.2, because rather than evaluating s in the state with $\mathcal{M}(S)$ applied to the entire variant, $\mathcal{M}(S)$ is applied to just the state $\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}$ and the variant $\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\}$ is then applied to the resulting state.

To see this, first let $\mathcal{M}(S)(\sigma) = \sigma\{\alpha_1/\xi_1\}\ldots\{\alpha_n/\xi_n\}$. Then it follows:

$$\begin{aligned}
&\mathcal{R}(s)((\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\}) \\
&\quad = \mathcal{R}(s)(((\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})\{\alpha_1/\xi_1\}\ldots\{\alpha_n/\xi_n\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\}) \\
&\qquad\qquad \text{(From above)} \\
&\quad = \mathcal{R}(s)((\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\}) \\
&\qquad\qquad \text{(Repeated application of Lemma 3.8.1)}
\end{aligned}$$

The proof of interchange with substitution then follows.

$$\begin{aligned}
&\mathcal{M}(x := s; S)(\sigma) \\
&\quad = \mathcal{M}(S)(\mathcal{M}(x := s)(\sigma)) && \text{(Def. of } S_1; S_2 \text{ [3.5.3])} \\
&\quad = \mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}) && \text{(Def. of } v := s \text{ [3.5.3])} \\
&\quad = (\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})) \\
&\qquad \{\mathcal{R}(x)(\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})) \\
&\qquad\quad /\mathcal{L}(x)(\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\} \\
&\qquad\qquad \text{(Sem. of state variants [3.6.3])} \\
&\quad = (\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})) \\
&\qquad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(x)(\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\} \\
&\qquad\qquad \text{(Def. of } \textit{sets} \text{ and } \textit{uses} \text{ [3.8.1])}
\end{aligned}$$

$= (\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))$
$\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\mathcal{M}(S)((\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})\})$
(Sem. of state variants [3.6.2])
$= (\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}$
(Def. of location of Svar [3.2.1])
$= (\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(s)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(x)(\sigma)\}$
(Sem. of state variants [3.6.3])
$= (\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))$
$\{\mathcal{R}(s)((\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(x)(\sigma)\}$
(Introduction of a variant [3.6.4])
$= (\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))$
$\{\mathcal{R}(s)((\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(x)(\sigma)\}$
(Preliminary result)
$= ((\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})$
$\{\mathcal{R}(s)((\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})/\mathcal{L}(x)(\sigma)\}$
(Introduction of a variant [3.6.4])
$= ((\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})$
$\{\mathcal{R}(s)((\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})$
$/\mathcal{L}(x)((\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})\}$
(Def. of location of Svar [3.2.1])
$= \mathcal{M}(x := s)((\mathcal{M}(S)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}))\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\}\ )$
(Def. of $v := s$ [3.5.3])
$= (\mathcal{M}(x := s)((\mathcal{M}(S[s/x])(\sigma))$ (Subst. into stat [3.7.3])
$= (\mathcal{M}(S[s/x]; x := s)(\sigma)$ (Def. of $S_1; S_2$ [3.5.3])

□

Notice that statement interchange with substitution provides the same result as copy propagation when the statement being interchanged is an assignment of the form $x := v$.

Yet another version of interchange with substitution, one in which the second is the assignment statement being used for the substitution, can be useful, particularly in backward copy statement propagation. Interchanging statements, propagating a copy statement backwards, rather than forwards, is discussed in the lemma below.

Theorem 4.1.5 (Interchange with backward substitution)

$\models v_1 := s; v_2 := x \quad = \quad v_2 := x; v_1[v_2/x] := s[v_2/x]; x := v_2$

Provided:

1) $\text{uses}(\mathcal{M}(v_1 := s)) \cap \text{sets}(\mathcal{M}(v_2 := x)) = \phi$

2) v_2 *is location-independent of* v_1.

3) v_2 *is strictly non-self-referencing.*

Proof: This can be proved by manipulation of the meaning of $v_1 := s; v_2 := x$. There will be two separate cases, either $v_1 \equiv x$ or it is not. Again, the cases in which either $v_1 := s$ or $v_2 := x$ are nullable follow directly and are not considered in the proof.

Case 1: $v_1 \equiv x$

$\mathcal{M}(v_1 := s; v_2 := x)(\sigma)$

$= \mathcal{M}(v_2 := x)(\mathcal{M}(v_1 := s)(\sigma))$ (Def. of $S_1; S_2$ [3.5.3])

$= \mathcal{M}(v_2 := x)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})$ (Def. of $v := s$ [3.5.3])

$= (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})$
$\{\mathcal{R}(x)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})/\mathcal{L}(v_2)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\}$
(Def. of $v := s$ [3.5.3])

$= (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})$
$\{\mathcal{R}(v_1)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})/\mathcal{L}(v_2)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\}$
(Given)

$= (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_2)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\}$
(Sem. of state variants [3.6.2])

$= (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_2)(\sigma)\}$
(Given)

$= (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\}$
(Interchange of state variants [3.6.1])

$= (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(v_2)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma)\}$
(Sem. of state variants [3.6.2])

$= (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_2)(\sigma)\})$
$\{\mathcal{R}(v_2)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\}$
(Def. of location of Svar [3.2.1])

$= \mathcal{M}(x := v_2)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_2)(\sigma)\})$ (Def. of $v := s$ [3.5.3])

$= \mathcal{M}(x := v_2)((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_2)(\sigma)\})$
(Introduction of a variant [3.6.4])

$= \mathcal{M}(x := v_2)$
$((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(v_2)(\sigma)\})$
(Given)

$= \mathcal{M}(x := v_2)((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})$
$\{\mathcal{R}(s)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(v_2)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})$
(Given)

$= \mathcal{M}(x := v_2)(\mathcal{M}(v_2 := s)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\}))$
(Def. of $v := s$ [3.5.3])

$= \mathcal{M}(x := v_2)(\mathcal{M}(v_2 := s)((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})$
(Sem. of state variants [3.6.3])
$= \mathcal{M}(x := v_2)(\mathcal{M}(v_2 := s)((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})$
(Interchange of state variants [3.6.1])
$= \mathcal{M}(x := v_2)(\mathcal{M}(v_2 := s)$
$((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(v_2)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma)\})$
(Sem. of state variants [3.6.2])
$= \mathcal{M}(x := v_2)(\mathcal{M}(v_2 := s)((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})$
$\{\mathcal{R}(v_2)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})$
(Def. of location of Svar [3.2.1])
$= \mathcal{M}(x := v_2)(\mathcal{M}(v_1[v_2/x] := s[v_2/x])(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\}))$
(Strong subst. into statements [3.7.4])
$= \mathcal{M}(v_1[v_2/x] := s[v_2/x]; x := v_2)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\}))$
(Def. of $S_1; S_2$ [3.5.3])
$= \mathcal{M}(v_1[v_2/x] := s[v_2/x]; x := v_2)(\mathcal{M}(v_2 := x)(\sigma))$
(Def. of $v := s$ [3.5.3])
$= \mathcal{M}(v_2 := x; v_1[v_2/x] := s[v_2/x]; x := v_2)(\sigma)$
(Def. of $S_1; S_2$ [3.5.3])

Case 2: $v_1 \not\equiv x$
As a preliminary result, notice that $\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v)(\sigma)\}\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\} = \sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v)(\sigma)\}$. This can be shown by looking at the two possible values of $\mathcal{L}(v)(\sigma)$. If $\mathcal{L}(v)(\sigma) = \mathcal{L}(x)(\sigma)$, this is just a case of a redundant variant and follows from Lemma 3.6.4. If $\mathcal{L}(v)(\sigma) \neq \mathcal{L}(x)(\sigma)$, this follows from interchange of the variants and elimination of the $\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\}$ variant, by Lemma 3.6.3.

A second preliminary result that is used in the following proof is that $\mathcal{R}(x)(\sigma) = \mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v)(\sigma)\})$. If $x \equiv v$, this follows by Lemma 3.6.3. If $x \not\equiv v$, this is a direct result of Lemma 3.6.2.

$\mathcal{M}(v_1 := s; v_2 := x)(\sigma)$
$= \mathcal{M}(v_2 := x)(\mathcal{M}(v_1 := s)(\sigma))$ (Def. of $S_1; S_2$ [3.5.3])
$= \mathcal{M}(v_2 := x)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})$ (Def. of $v := s$ [3.5.3])
$= (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})$
$\{\mathcal{R}(x)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})/\mathcal{L}(v_2)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\}$
(Def. of $v := s$ [3.5.3])
$= (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})$
$\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\}$
(Sem. of state variants [3.6.2])
$= (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\}$
(Given)
$= (\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}$
(Interchange of state variants [3.6.1])
$= ((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})$
$\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}$ (First preliminary result)

$$
\begin{aligned}
&= ((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}) \\
&\quad \{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\} \qquad \text{(Interchange of state variants [3.6.1])} \\
&= (((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\}) \\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\}) \\
&\quad\quad \{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\} \\
&\qquad \text{(Sem. of state variants [3.6.3])} \\
&= (((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\}) \\
&\quad\quad \{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\} \\
&\qquad \text{(Second preliminary result)} \\
&= (((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}) \\
&\quad \{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\} \\
&\qquad \text{(Interchange of state variants [3.6.1])} \\
&= (((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}) \\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\}) \\
&\quad\quad \{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\} \qquad \text{(Second preliminary result)} \\
&= (((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}) \\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\}) \\
&\quad\quad \{\mathcal{R}(v_2)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma)\} \\
&\qquad \text{(Sem. of state variants [3.6.2])} \\
&= (((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}) \\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})/\mathcal{L}(v_2)(\sigma)\}) \\
&\quad\quad \{\mathcal{R}(v_2)((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})/\mathcal{L}(x)(\sigma)\} \\
&\qquad \text{(Sem. of state variants [3.6.2])} \\
&= (((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}) \\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})/\mathcal{L}(v_2)(\sigma)\}) \\
&\quad\quad \{\mathcal{R}(v_2)(((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}) \\
&\quad\quad\quad \{\mathcal{R}(x)((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\})/\mathcal{L}(x)(\sigma)\} \\
&\quad\quad\quad\quad /\mathcal{L}(x)((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\})/\mathcal{L}(x)(\sigma)\} \\
&\qquad \text{(Sem. of state variants [3.6.3])} \\
&= (((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}) \\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})/\mathcal{L}(v_2)(\sigma)\}) \\
&\quad\quad \{\mathcal{R}(v_2)(((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}) \\
&\quad\quad\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\}) \\
&\quad\quad\quad\quad /\mathcal{L}(x)((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\})/\mathcal{L}(x)(\sigma)\} \\
&\qquad \text{(Sem. of state variants [3.6.2])} \\
&= (((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}) \\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})/\mathcal{L}(v_2)(\sigma)\}) \\
&\quad\quad \{\mathcal{R}(v_2)(((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\}) \\
&\quad\quad\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\}) \\
&\quad\quad\quad /\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})/\mathcal{L}(x)(\sigma)\} \\
&\qquad\qquad \text{(Def. of location of Svar [3.2.1])}
\end{aligned}
$$

$$
\begin{aligned}
&= ((((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})/\mathcal{L}(v_2)(\sigma)\})\\
&\qquad \{\mathcal{R}(v_2)(((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\\
&\qquad\qquad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad /\mathcal{L}(x)(((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\\
&\qquad\qquad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\}\\
&\qquad\qquad\qquad \text{(Def. of location of Svar [3.2.1])}\\
&= \mathcal{M}(x := v_2)(((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma)/\mathcal{L}(v_1)(\sigma)\})\\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad\qquad\qquad \text{(Def. of } v := s \text{ [3.5.3])}\\
&= \mathcal{M}(x := v_2)(((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\mathcal{M}(v_2 := x)(\sigma))/\mathcal{L}(v_1)(\sigma)\})\\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad\qquad\qquad \text{(Def. of empty } \textit{sets} \text{ and } \textit{uses} \text{ [3.8.2])}\\
&= \mathcal{M}(x := v_2)(((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(s)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(v_1)(\sigma)\})\\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad\qquad\qquad \text{(Def. of } v := s \text{ [3.5.3])}\\
&= \mathcal{M}(x := v_2)(((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\\
&\quad \{\mathcal{R}(s)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(v_1)(\mathcal{M}(v_2 := x)(\sigma))\})\\
&\qquad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad\qquad\qquad \text{(Def. of empty } \textit{sets} \text{ and } \textit{uses} \text{ [3.8.2])}\\
&= \mathcal{M}(x := v_2)(((\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\\
&\quad \{\mathcal{R}(s)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(v_1)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad\qquad\qquad \text{(Def. of } v := s \text{ [3.5.3])}\\
&= \mathcal{M}(x := v_2)(((\mathcal{M}(v_1 := s)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad\qquad\qquad \text{(Def. of } v := s \text{ [3.5.3])}\\
&= \mathcal{M}(x := v_2)(((\mathcal{M}(v_1 := s)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\{\mathcal{R}(x)(\sigma)/\mathcal{L}(x)(\sigma)\})\\
&\quad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad\qquad\qquad \text{(First preliminary result)}\\
&= \mathcal{M}(x := v_2)(((\mathcal{M}(v_1 := s)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\\
&\quad \{\mathcal{R}(v_2)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma)\})\\
&\qquad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad\qquad\qquad \text{(Sem. of state variants [3.6.2])}\\
&= \mathcal{M}(x := v_2)(((\mathcal{M}(v_1 := s)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\\
&\quad \{\mathcal{R}(v_2)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad \{\mathcal{R}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})/\mathcal{L}(x)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\})\\
&\qquad\qquad\qquad \text{(Def. of location of Svar [3.2.1])}\\
&= \mathcal{M}(x := v_2)(\mathcal{M}(v_1[v_2/x] := s[v_2/x])(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\\
&\qquad\qquad\qquad \text{(Strong subst. into statements [3.7.4])}\\
&= \mathcal{M}(v_1[v_2/x] := s[v_2/x]; x := v_2)(\sigma\{\mathcal{R}(x)(\sigma)/\mathcal{L}(v_2)(\sigma)\})\\
&\qquad\qquad\qquad \text{(Def. of } S_1; S_2 \text{ [3.5.3])}
\end{aligned}
$$

$$= \mathcal{M}(v_1[v_2/x] := s[v_2/x]; x := v_2)(\mathcal{M}(v_2 := x)(\sigma))$$
(Def. of $v := s$ [3.5.3])
$$= \mathcal{M}(v_2 := x; v_1[v_2/x] := s[v_2/x]; x := v_2)(\sigma)$$
(Def. of $S_1; S_2$ [3.5.3])

□

Compression is used to remove statements from the code when they are no longer needed. A pair of statements will compress down to the second statement if the meaning of the second is the same as the meaning of the pair. (This may be combined with statement interchange to provide compression to the first statement.)

Definition 4.1.2 (Compressible statements)

A pair of statements $S_1; S_2$ is compressible *(to S_2) in state σ if $\mathcal{M}(S_1; S_2)(\sigma) = \mathcal{M}(S_2)(\sigma)$.*

The simplest cases for statement compression follow from the conditions below. More complex compression can take place by first applying the other transformations (such as absorption into `if` statements) to create these conditions. This will be discussed in more detail in chapter 5.

Theorem 4.1.6 (Compression of statements)

The following are sufficient for the compression of $S_1; S_2$:

a) $S_1 = v := s_1$ and $S_2 = v := s_2$ when sets$(v := s_1) \cap$ uses$(v := s_2) = \phi$.

b) S_1 is nullable.

Proof:
The proof of part b) follows directly from the definition of nullable statements. The proof of part a) is provided below.

$$\mathcal{M}(v := s_1; v := s_2)(\sigma)$$
$$= \mathcal{M}(v := s_2)(\mathcal{M}(v := s_1)(\sigma)) \quad \text{(Def. of } S_1; S_2 \text{ [3.5.3])}$$
$$= (\mathcal{M}(v := s_1)(\sigma))\{\mathcal{R}(s_2)(\mathcal{M}(v := s_1)(\sigma))/\mathcal{L}(v)(\mathcal{M}(v := s_1)(\sigma))\}$$
(Def. of $v := s$ [3.5.3])
$$= (\mathcal{M}(v := s_1)(\sigma))\{\mathcal{R}(s_2)(\sigma)/\mathcal{L}(v)(\sigma)\} \quad \text{(Def. of empty } \textit{sets} \text{ and } \textit{uses} \text{ [3.8.2])}$$

$= (\sigma\{\mathcal{R}(s_1)(\sigma)/\mathcal{L}(v)(\sigma)\})\{\mathcal{R}(s_2)(\sigma)/\mathcal{L}(v)(\sigma)\}$
(Def. of $v := s$ [3.5.3])
$= \sigma\{\mathcal{R}(s_2)(\sigma)/\mathcal{L}(v)(\sigma)\}$ (Elimination of state variant [3.6.4])
$= \mathcal{M}(v := s_2)(\sigma)$ (Def. of $v := s$ [3.5.3])

□

4.2 Primitive `if` Statement Transformations

There are a variety of low-level `if` statement transformations. The first, absorption of statements into `if` statements, moves statements from either before or after the `if` statement into both the `then` and `else` clauses of the `if` statement (or conversely, moves statements out of those clauses). Bottom factoring is a widely recognized method of `if` statement simplification. Statements following an `if` statement may always be moved into the clauses. The case for top factoring is somewhat more complex. If absorption is discussed in the following theorems.

Theorem 4.2.1 (Absorption of statements into `if` *statements)*

$\models$ *S;* `if` *b* `then` S_1 `else` S_2 `fi` =

`if` *b* `then` *S;*S_1 `else` *S;*S_2 `fi;`

Provided:

sets$(\mathcal{M}(S))$ ∩ uses$(b) = \phi$.

and

$\models$ `if` *b* `then` S_1 `else` S_2 `fi;` S = `if` *b* `then` $S_1; S;$ `else` $S_2; S$ `fi`

Proof of the first part:

$\mathcal{M}(S;$ `if` *b* `then` S_1 `else` S_2 `fi`$)(\sigma)$
$= \mathcal{M}($`if` *b* `then` S_1 `else` S_2 `fi`$)\ (\mathcal{M}(S)(\sigma))$
(Def. of $S_1; S_2$ [3.5.3])
$=$ *if* $\mathcal{W}(b)(\mathcal{M}(S)(\sigma))$ *then* $\mathcal{M}(S_1)(\mathcal{M}(S)(\sigma))$
else $\mathcal{M}(S_2)(\mathcal{M}(S)(\sigma))$ *fi* (Def. of `if` [3.5.3])
$=$ *if* $\mathcal{W}(b)(\sigma)$ *then* $\mathcal{M}(S_1)(\mathcal{M}(S)(\sigma))$
else $\mathcal{M}(S_2)(\mathcal{M}(S)(\sigma))$ *fi* (Def. of empty *sets* and *uses* [3.8.3])

$= \mathit{if}\ \mathcal{W}(b)(\sigma)\ \mathit{then}\ \mathcal{M}(S;S_1)(\sigma)\ \mathit{else}\ \mathcal{M}(S;S_2)(\sigma)\ \mathit{fi}$
(Def. of $S_1; S_2$ [3.5.3])
$= \mathcal{M}(\texttt{if}\ b\ \texttt{then}\ S; S_1\ \texttt{else}\ S; S_2\ \texttt{fi})(\sigma)$ (Def. of if [3.5.3])

Proof of the second part:
$\mathcal{M}(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi};\ S)(\sigma)$
$= \mathcal{M}(S)(\mathcal{M}(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi})(\sigma))$
(Def. of $S_1; S_2$ [3.5.3])
$= \mathcal{M}(S)(\mathit{if}\ \mathcal{W}(b)(\sigma)\ \mathit{then}\ \mathcal{M}(S_1)(\sigma)\ \mathit{else}\ \mathcal{M}(S_2)(\sigma)\ \mathit{fi})$
(Def. of if [3.5.3])
$= \mathit{if}\ \mathcal{W}(b)(\sigma)\ \mathit{then}\ \mathcal{M}(S)(\mathcal{M}(S_1)(\sigma))\ \mathit{else}\ \mathcal{M}(S)(\mathcal{M}(S_2)(\sigma))\ \mathit{fi}$
(Moving $\mathcal{M}(S)$ into both branches)
$= \mathit{if}\ \mathcal{W}(b)(\sigma)\ \mathit{then}\ \mathcal{M}(S_1;S)(\sigma)$
$\mathit{else}\ \mathcal{M}(S_2;S)(\sigma)\ \mathit{fi}$ (Def. of $S_1; S_2$ [3.5.3])
$= \mathcal{M}(\texttt{if}\ b\ \texttt{then}\ S_1;S\ \texttt{else}\ S_2;S\ \texttt{fi})(\sigma)$ (Def. of if[3.5.3])
□

In some cases it may be desirable to move statements from before an `if` statement into the `if` statement even if those statements possibly change the meaning of the condition. Any assignment statement which assigns to a simple variable can be moved into an `if` statement, if there is substitution performed in the condition.

Theorem 4.2.2 (Absorption of statements into `if` *statements with substitution)*

$\models x := s;\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi} \quad = \quad \texttt{if}\ b[s/x]\ \texttt{then}\ x := s;\ S_1\ \texttt{else}\ x := s;\ S_2\ \texttt{fi}$

Proof: This can be proved by manipulation of the meaning of $x := s$; `if` b `then` S_1 `else` S_2 `fi`.

$\mathcal{M}(x := s;\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi})\ (\sigma)$
$= \mathcal{M}(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi})\ (\mathcal{M}(x := s)(\sigma))$
(Def. of $S_1; S_2$ [3.5.3])
$= \mathcal{M}(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi})\ (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})$
(Def. of $v := s$ [3.5.3])
$= \mathit{if}\ \mathcal{W}(b)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})\ \mathit{then}\ \mathcal{M}(S_1)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})$
$\mathit{else}\ \mathcal{M}(S_2)\ (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})\ \mathit{fi}$ (Def. of if [3.5.3])

$= \mathit{if}\ \mathcal{W}(b[s/x])(\sigma)\ \mathit{then}\ \mathcal{M}(S_1)(\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})$
$\quad \mathit{else}\ \mathcal{M}(S_2)\ (\sigma\{\mathcal{R}(s)(\sigma)/\mathcal{L}(x)(\sigma)\})\ \mathit{fi}$ (Lemma 3.7.1)
$= \mathit{if}\ \mathcal{W}(b[s/x])(\sigma)\ \mathit{then}\ \mathcal{M}(S_1)(\mathcal{M}(x := s)(\sigma))$
$\quad \mathit{else}\ \mathcal{M}(S_2)(\mathcal{M}(x := s)(\sigma))\ \mathit{fi}$ (Def. of $v := s$ [3.5.3])
$= \mathit{if}\ \mathcal{W}(b[s/x])(\sigma)\ \mathit{then}\ \mathcal{M}(x := s; S_1)(\sigma)$
$\quad \mathit{else}\ \mathcal{M}(x := s;\ S_2)(\sigma)\ \mathit{fi}$ (Def. of $S_1; S_2$ [3.5.3])
$= \mathcal{M}(\texttt{if}\ b[s/x]\ \texttt{then}\ x := s;\ S_1\ \texttt{else}\ x := s;\ S_2\ \texttt{fi})\ (\sigma)$
(Def. of if [3.5.3])

□

In some cases, the **else** clause of an **if then else** statement is empty, resulting in an **if then** statement. In order to simplify notation, both the **if then** statement and max and min functions are defined below.

*Definition 4.2.1 (*max *and* min*)*

$max(m, n) =$ `if` $m > n$ `then` m `else` n `fi`

$min(m, n) =$ `if` $m < n$ `then` m `else` n `fi`

Definition 4.2.2 (`if then` *statement)*

`if` b `then` S_1 `fi` = `if` b `then` S_1 `else` D `fi`

Since later optimizations only work with **if then** statements (as opposed to **if then else** statements), it may be necessary to transform an **if then else** statement into two **if then** statements. This is done in the following theorem.

Theorem 4.2.3 (Splitting `if then else` *statements)*

$\models$`if` b `then` S_1 `else` S_2 `fi` = `if` b `then` S_1 `fi;` `if` $\neg b$ `then` S_2 `fi`

Provided:

$\mathrm{sets}(\mathcal{M}(S_1)) \cap \mathrm{uses}(b) = \phi.$

Proof:

$\mathcal{M}$(`if` b `then` S_1 `else` S_2 `fi`)(σ)

$= \mathit{if}\ \mathcal{W}(b)(\sigma)\ \mathit{then}\ \mathcal{M}(S_1)(\sigma)\ \mathit{else}\ \mathcal{M}(S_2)(\sigma)\ \mathit{fi}$
(Def. of `if` [3.5.3])

$= \mathit{if}\ \neg\mathcal{W}(b)(\sigma)\ \mathit{then}\ \mathcal{M}(S_2)(\sigma)\ \mathit{else}\ \mathcal{M}(S_1)(\sigma)\ \mathit{fi}$
(Meaning of if)

$= \mathit{if}\ \mathcal{W}(\neg b)(\sigma)\ \mathit{then}\ \mathcal{M}(S_2)(\sigma)\ \mathit{else}\ \mathcal{M}(S_1)(\sigma)\ \mathit{fi}$
(Def. of $\mathcal{W}(\neg b)$ [3.5.2])

$= \mathit{if}\ \mathcal{W}(\neg b)(\sigma)\ \mathit{then}\ \mathcal{M}(S_2)(\mathcal{M}(D)(\sigma))\ \mathit{else}\ \mathcal{M}(D)(\mathcal{M}(S_1)(\sigma))\ \mathit{fi}$
(Def. of D [3.5.3])

$= \mathit{if}\ \mathcal{W}(\neg b)(\sigma)\ \mathit{then}$
$\quad \mathit{if}\ \mathcal{W}(b)(\sigma)\ \mathit{then}\ \mathcal{M}(S_2)(\mathcal{M}(S_1)(\sigma))\ \mathit{else}\ \mathcal{M}(S_2)(\mathcal{M}(D)(\sigma))\ \mathit{fi}$
$\quad \mathit{else if}\ \mathcal{W}(b)(\sigma)\ \mathit{then}\ \mathcal{M}(D)(\mathcal{M}(S_1)(\sigma))$
$\quad\quad \mathit{else}\ \mathcal{M}(D)(\mathcal{M}(D)(\sigma))\ \mathit{fi}\ \mathit{fi}$ (Vacuously true)

$= \mathit{if}\ \mathcal{W}(\neg b)(\sigma)\ \mathit{then}$
$\quad \mathcal{M}(S_2)(\mathit{if}\ \mathcal{W}(b)(\sigma)\ \mathit{then}\ \mathcal{M}(S_1)(\sigma)\ \mathit{else}\ \mathcal{M}(D)(\sigma)\ \mathit{fi})$
$\quad \mathit{else}\ \mathcal{M}(D)(\mathit{if}\ \mathcal{W}(b)(\sigma)\ \mathit{then}\ \mathcal{M}(S_1)(\sigma)\ \mathit{else}\ \mathcal{M}(D)(\sigma)\ \mathit{fi})\ \mathit{fi}$
(Removing $\mathcal{M}(S_2)$ from both of the branches in the first clause and $\mathcal{M}(D)$ from both branches in the second clause.)

$= \mathit{if}\ \mathcal{W}(\neg b)(\sigma)\ \mathit{then}$
$\quad \mathcal{M}(S_2)(\mathcal{M}($`if` b `then` S_1 `else` D `fi`$)(\sigma))$
$\quad \mathit{else}\ \mathcal{M}(D)(\mathcal{M}($`if` b `then` S_1 `else` D `fi`$)(\sigma))$
$\quad \mathit{fi}$
(Def. of `if` [3.5.3])

$= \mathit{if}\ \mathcal{W}(\neg b)(\sigma)\ \mathit{then}$
$\quad \mathcal{M}(S_2)(\mathcal{M}($`if` b `then` S_1 `fi`$)(\sigma))$
$\quad \mathit{else}\ \mathcal{M}(D)(\mathcal{M}($`if` b `then` S_1 `fi`$)(\sigma))\ \mathit{fi}$
(Def. of `if then fi` [4.2.2])

$= \mathit{if}\ \mathcal{W}(\neg b)(\mathcal{M}($`if` b `then` S_1 `fi`$)(\sigma))\ \mathit{then}$
$\quad \mathcal{M}(S_2)(\mathcal{M}($`if` b `then` S_1 `fi`$)(\sigma))$
$\quad \mathit{else}\ \mathcal{M}(D)(\mathcal{M}($`if` b `then` S_1 `fi`$)(\sigma))\ \mathit{fi}$
(Def. of empty *sets* and *uses* [3.8.3])

$= \mathcal{M}($`if` $\neg b$ `then` S_2 `else` D `fi`$)\ (\mathcal{M}($`if` b `then` S_1 `fi`$)(\sigma))$
(Def. of `if` [3.5.3])

$= \mathcal{M}($`if` $\neg b$ `then` S_2 `fi`$)\ (\mathcal{M}($`if` b `then` S_1 `fi`$)(\sigma))$
(Def. of `if then fi` [4.2.2])

$= \mathcal{M}($`if` b `then` S_1 `fi;` `if` $\neg b$ `then` S_2 `fi`$)(\sigma)$
(Def. of $S_1; S_2$ [3.5.3])

□

Corollary 4.2.4 (Splitting `if then else` *Statements)*

$\models$`if` b `then` S_1 `else` S_2 `fi` = `if` $\neg b$ `then` S_2 `fi;` `if` b `then` S_1 `fi`

Provided:

$\text{sets}(\mathcal{M}(S_2)) \cap \text{uses}(b) = \phi$

This is proved in the same way as Theorem 4.2.3.

Finally, some `if` statements can be simplified to single simpler statements. If the truth value of a condition is the same in all states, an `if` statement can be simplified to either the **then** or **else** clause. If both the **then** and the **else** clauses of an `if` statement contain nullable statements, then the `if` statement can become the empty statement. This result shows an interesting result of the fact that no effects of errors are considered during evaluation of expressions. Consider the statement `if` $x/0 = 4$ `then` D `else` D `fi`. If this were reduced to simply D, the meaning of the code would be vastly different—the first case would cause a run time error and the second would have no such error. The third clause, combined with `if` statement extraction (Lemma 4.2.1), allow the simplification of statements of the form `if` b `then` S `else` S `fi`.

Theorem 4.2.5 (`if` *simplification)*

$\models$ (`if` b `then` S_1 `else` S_2 `fi`) = S_1

Provided: $\mathcal{W}(b)(\sigma) = \mathbf{T}$ *for all* $\sigma \in \Sigma$

$\models$ (`if` b `then` S_1 `else` S_2 `fi`) = S_2

Provided: $\mathcal{W}(b)(\sigma) = \mathbf{F}$ *for all* $\sigma \in \Sigma$

$\models$ (`if` b `then` S_1 `else` S_2 `fi`) = D

Provided: S_1 *and* S_2 *are nullable.*

Proof of the first part:

$\mathcal{M}$(**if** b **then** S_1 **else** S_2 **fi**)(σ)

= *if* $\mathcal{W}(b)(\sigma)$ *then* $\mathcal{M}(S_1)(\sigma)$ *else* $\mathcal{M}(S_2)(\sigma)$ *fi* (Def. of **if** [3.5.3])

= $\mathcal{M}(S_1)(\sigma)$ (Given)

Proof of the second part:

$\mathcal{M}$(**if** b **then** S_1 **else** S_2 **fi**)(σ)

= *if* $\mathcal{W}(b)(\sigma)$ *then* $\mathcal{M}(S_1)(\sigma)$ *else* $\mathcal{M}(S_2)(\sigma)$ *fi* (Def. of **if** [3.5.3])

= $\mathcal{M}(S_2)(\sigma)$ (Given)

Proof of the third part:

$\mathcal{M}$(**if** b **then** S_1 **else** S_2 **fi**)(σ)

= *if* $\mathcal{W}(b)(\sigma)$ *then* $\mathcal{M}(S_1)(\sigma)$ *else* $\mathcal{M}(S_2)(\sigma)$ *fi* (Def. of **if** [3.5.3])

= *if* $\mathcal{W}(b)(\sigma)$ *then* $\mathcal{M}(D)(\sigma)$ *else* $\mathcal{M}(D)(\sigma)$ *fi* (Given)

= $\mathcal{M}(D)(\sigma)$ (Meaning of if)

□

4.3 Primitive Loop Transformation

There is only one basic transformation needed for loop statements which execute at least once, the unrolling of the **for**, removing either the first or last iteration of the statement from the loop. If the loop does not execute at least once, the only transformation necessary is changing it into the empty statement. All subsequent **for** statement transformations will be proved by viewing the transformation as if it unrolls the **for** statement entirely, performs the transformation, and then rerolls the loop. The most complicated case of this is unrolling a statement to before the **for** statement. This is discussed in the second part of Theorem 4.3.2.

Theorem 4.3.1 (Loop elimination)

$\mathcal{M}$(**for** $x := s_1$ **to** s_2 **do** S **od**)(σ) =

$= \sigma$

if $\mathcal{R}(s_1)(\sigma) \not\leq \mathcal{R}(s_2)(\sigma)$

This proof follows directly from the definition of the semantics of **for** statements (Definition 3.5.3).

Theorem 4.3.2 (Loop rolling and unrolling)

$\models \mathcal{M}$(**for** $x := s_1$ **to** s_2 **do** S **od**)(σ)

$= \mathcal{M}$(**for** $x := s_1$ **to** $s_2 - 1$ **do** S **od**; $S[\overline{\mathcal{R}(s_2)(\sigma)}/x]$)$(\sigma)$

$= \mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x]$; **for** $x := \overline{\mathcal{R}(s_1)(\sigma)} + 1$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)(\sigma)$

if $\mathcal{R}(s_1)(\sigma) \leq \mathcal{R}(s_2)(\sigma)$

The first two parts are equal by the definition of **for** statements (Definition 3.5.3). The proof of the equivalence of the first and third part is by mathematical induction on the number of times through the loop ($\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1$) and follows.

Basis: $\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1 = 1$

$\mathcal{M}$(**for** $x := s_1$ **to** s_2 **do** S **od**)(σ)

$= \mathcal{M}$(**for** $x := s_1$ **to** $s_2 - 1$ **do** S **od**; $S[\overline{\mathcal{R}(s_2)(\sigma)}/x]$)$(\sigma)$ (Def. of **for** [3.5.3])

$= \mathcal{M}(D; S[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$ (Def. of **for** [3.5.3])

$= \mathcal{M}(S[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$ (Def. of D [3.5.3])

$= \mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma)$ (Given)

$= \mathcal{M}(D)(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$ (Def. of D [3.5.3])

$= \mathcal{M}$(**for** $x := \overline{\mathcal{R}(s_1)(\sigma)} + 1$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**)
$(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$ (Def. of **for** [3.5.3])

$= \mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x]$; **for** $x := \overline{\mathcal{R}(s_1)(\sigma)} + 1$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)(\sigma)$
(Def. of $S_1; S_2$ [3.5.3])

Induction step: Assume that **for** $x := s_1$ **to** s_2 **do** S **od** $= S[\overline{\mathcal{R}(s_1)(\sigma)}/x]$; **for** $x := \overline{\mathcal{R}(s_1)(\sigma)} + 1$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od** whenever $\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1 = k$ ($k \geq 1$).

Show it is true when $\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1 = \mathrm{k} + 1$.

$\mathcal{M}(\texttt{for}\ x := s_1\ \texttt{to}\ s_2\ \texttt{do}\ S\ \texttt{od})(\sigma)$

$= \mathcal{M}(\texttt{for}\ x := s_1\ \texttt{to}\ s_2 - 1\ \texttt{do}\ S\ \texttt{od};\ S[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$
(Def. of `for` [3.5.3])

$= \mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x];$
$\texttt{for}\ x := \overline{\mathcal{R}(s_1)(\sigma)} + 1\ \texttt{to}\ \overline{\mathcal{R}(s_2 - 1)(\sigma)}\ \texttt{do}\ S\ \texttt{od};\ S[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$
(Induction hypothesis)

$= \mathcal{M}(\texttt{for}\ x := \overline{\mathcal{R}(s_1)(\sigma)} + 1\ \texttt{to}\ \overline{\mathcal{R}(s_2 - 1)(\sigma)}\ \texttt{do}\ S\ \texttt{od};\ S[\overline{\mathcal{R}(s_2)(\sigma)}/x])$
$(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$
(Def. of $S_1; S_2$ [3.5.3])

$= \mathcal{M}(\texttt{for}\ x := \overline{\mathcal{R}(s_1)(\sigma)} + 1\ \texttt{to}\ \overline{\mathcal{R}(s_2)(\sigma) - 1}\ \texttt{do}\ S\ \texttt{od};\ S[\overline{\mathcal{R}(s_2)(\sigma)}/x])$
$(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$
(Def. of $\mathcal{R}(s-1)$ [3.5.1])

$= \mathcal{M}(\texttt{for}\ x := \overline{\mathcal{R}(\overline{\mathcal{R}(s_1)(\sigma)} + 1)(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))}$
$\texttt{to}\ \overline{\mathcal{R}(\overline{\mathcal{R}(s_2)(\sigma)} - 1)(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))}\ \texttt{do}\ S\ \texttt{od};$
$S[\overline{\mathcal{R}(\overline{\mathcal{R}(s_2)(\sigma)})(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))}/x]\ (\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$
(Def. of $\mathcal{R}(m)$ [3.5.1])

$= \mathcal{M}(\texttt{for}\ x := \overline{\mathcal{R}(\overline{\mathcal{R}(s_1)(\sigma)})(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))} + 1$
$\texttt{to}\ \overline{\mathcal{R}(\overline{\mathcal{R}(s_2)(\sigma)})(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))} - 1\ \texttt{do}\ S\ \texttt{od};$
$S[\overline{\mathcal{R}(\overline{\mathcal{R}(s_2)(\sigma)})(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))}/x]\ (\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$
(Def. of $\mathcal{R}(s-1)$ [3.5.1])

$= \mathcal{M}(\texttt{for}\ x := \overline{\mathcal{R}(\overline{\mathcal{R}(s_1)(\sigma)})(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))} + 1$
$\texttt{to}\ \overline{\mathcal{R}(\overline{\mathcal{R}(s_2)(\sigma)})(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))}\ \texttt{do}\ S\ \texttt{od})$
$(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$ (Def. of `for` [3.5.3])

$= \mathcal{M}(\texttt{for}\ x := \overline{\mathcal{R}(s_1)(\sigma)} + 1\ \texttt{to}\ \overline{\mathcal{R}(s_2)(\sigma)}\ \texttt{do}\ S\ \texttt{od})$
$(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$ (Def. of $\mathcal{R}(m)$ [3.5.1])

$= \mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x];$
$\texttt{for}\ x := \overline{\mathcal{R}(s_1)(\sigma)} + 1\ \texttt{to}\ \overline{\mathcal{R}(s_2)(\sigma)}\ \texttt{do}\ S\ \texttt{od})\ (\sigma)$
(Def. of $S_1; S_2$ [3.5.3])

□

CHAPTER 5
GLOBAL OPTIMIZATIONS

The basic transformations discussed in Chapter 4 can be combined to give traditional data flow analysis optimizations and some new optimizations as well. Most of these transformations can be viewed as unrolling one or more loops, interchanging the resulting unrolled statements, possibly with some statement elimination, restructuring, or simplification (such as statement compression or **if then else** simplification) and rerolling of the code into a loop again.

Loop joining (Section 5.1) unrolls the statements in two adjacent loops, rearranges them, and rerolls them into a single loop. Similarly, loop interchanging (Section 5.2) requires no elimination or restructuring of the unrolled statements—the statements in a pair of nested loops are simply unrolled, rearranged, and then rerolled. In code motion (Section 5.3), after the loops are unrolled, the copies of the statement being removed from the loop are moved to the beginning of the unrolled statements. These statements are then compressed to a single statement and the remaining statements are rerolled, giving a single statement followed by a loop. Finally, loop-conditional joining (Section 5.4) unrolls the loop, simplifies the conditional statement which makes up the body of the loop (and in doing may remove some of the conditional statements), and then rerolls the remaining statements, which no longer have the original conditional clause.

Besides describing each complex transformation in terms of the minimal transformations, it is necessary to determine when the complex transformations can occur and when they will improve the code. In all but the smallest of programs, these

transformations must be attempted in some order. Heuristic methods for combining these transformations and the transformations presented in the previous chapter are presented in the context of a prototype optimizer in Section 6.4.

5.1 Loop Joining

Loop joining, that is, the combining of two (or more, by repeated application) loops that are executed over a similar range of loop boundaries, is another transformation which provides its greatest benefits in rearranging the grouping of statements. Loop joining has some benefits in eliminating the initialization of the second loop and consolidating the loop control variable increment and examination costs of two loops. Loop joining (and the reverse operation of loop splitting) can be part of more spectacular benefits when they are used to rearrange code so that other optimizations, most notably loop-conditional joining, can occur.

Theorem 5.1.1 (Loop Joining)

$\models \mathcal{M}($**for** $x := s_1$ **to** s_2 **do** S_1 **od**; **for** $y := s_1$ **to** s_2 **do** S_2 **od**$)(\sigma) =$

$\mathcal{M}($**for** $x := s_1$ **to** s_2 **do** $S_1; S_2[x/y]$ **od**$)(\sigma)$

Provided:

$\text{sets}(S_1) \cap \text{uses}(s_1) = \phi$

$\text{sets}(S_1) \cap \text{uses}(s_2) = \phi$

$\forall y_1, y_2 \ni \mathcal{R}(s_1)(\sigma) \leq y_1 < y_2 \leq \mathcal{R}(s_2)(\sigma), S_1[y_1/x]$ *is interchangeable with* $S_2[y_2/y]$

Proof: Proof by induction on the number of times through the loop, $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1)$.

Basis: $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) \leq 0$ (no iterations).

$\mathcal{M}($**for** $x := s_1$ **to** s_2 **do** S_1 **od**; **for** $y := s_1$ **to** s_2 **do** S_2 **od**$)(\sigma)$

$= \mathcal{M}(D;$ **for** $y := s_1$ **to** s_2 **do** S_2 **od**$)(\sigma)$ (Loop unrolling [4.3.1])

$= \mathcal{M}($**for** $y := s_1$ **to** s_2 **do** S_2 **od**$)(\sigma)$ (Def. of D [3.5.3])

$= \mathcal{M}(D)(\sigma)$ (Loop unrolling [4.3.1])

$= \mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ do } S_1; S_2[x/y] \texttt{ od})(\sigma)$
(Loop rolling [4.3.1])

Induction step: Assume that `for` $x := s_1$ `to` s_2 `do` S_1 `od; for` y := s_1 `to` s_2 `do` S_2 `od` = `for` $x := s_1$ `to` s_2 `do` $S_1; S_2[x/y]$ `od`; in states where $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) = k$ $(k \geq 0)$.

Show it is true in states where $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) = k + 1$.
$\mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ do } S_1 \texttt{ od; for } y := s_1 \texttt{ to } s_2 \texttt{ do } S_2 \texttt{ od})(\sigma)$
$= \mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ - 1 do } S_1 \texttt{ od; } S_1[\overline{\mathcal{R}(s_2)(\sigma)}/x];$
$\texttt{for } y := s_1 \texttt{ to } s_2 \texttt{ - 1 do } S_2 \texttt{ od; } S_2[\overline{\mathcal{R}(s_2)(\sigma)}/y])(\sigma)$
(Loop unrolling [4.3.2])
$= \mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ - 1 do } S_1 \texttt{ od; for } y := s_1 \texttt{ to } s_2 \texttt{ - 1 do } S_2 \texttt{ od;}$
$S_1[\overline{\mathcal{R}(s_2)(\sigma)}/x];\ S_2[\overline{\mathcal{R}(s_2)(\sigma)}/y])(\sigma)$ (Given)
$= \mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ - 1 do } S_1; S_2[x/y] \texttt{ od;}$
$S_1[\overline{\mathcal{R}(s_2)(\sigma)}/x];\ S_2[\overline{\mathcal{R}(s_2)(\sigma)}/y])(\sigma)$ (Induction hypothesis)
$= \mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ - 1 do } S_1; S_2[x/y] \texttt{ od;}$
$S_1[\overline{\mathcal{R}(s_2)(\sigma)}/x];\ S_2[x/y][\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$
(Subst. into stat. [3.4.2])
$= \mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ - 1 do } S_1; S_2[x/y] \texttt{ od;}$
$(S_1; S_2[x/y])[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$ (Def. of subst. into stat. [3.4.4])
$= \mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ do } S_1; S_2[x/y] \texttt{ od})(\sigma)$
(Loop rolling [4.3.2])

□

5.2 Loop Interchange

Loop interchanging is a code transformation which may not provide any immediate benefit to the code. It is used to change the order of the indices of two (or more, by repeated application) loops. This can be viewed as a change from row-major to column-major traversal of an array. (Alternatively, this may be viewed as transition from column-major to row-major—for the sake of consistency the row-major to column-major interpretation will be used here.) There may be a slight increase or decrease in the speed of the code because loop interchange will change the number of times the inner loop will be initialized and may change the number of times each of the loop conditions will be evaluated. While this change in loop overhead may

```
for x1 := 1 to 1000 do
   for x2 := 1 to 2 do
      S
   od
od
```

Outer loop initialization	1
Inner loop initialization	1000
Outer loop condition check	1001
Inner loop condition check	3000

```
for x2 := 1 to 2 do
   for x1 := 1 to 1000 do
      S
   od
od
```

Outer loop initialization	1
Inner loop initialization	2
Outer loop condition check	3
Inner loop condition check	2002

Figure 5.1. Loop Interchange May Change the Number of Loop Initializations

seem to be reason enough for loop interchanging in extreme conditions such as the one in Figure 5.1, it is usually insignificant when compared to the real power of loop interchanging.

The major advantage of loop joining is that it allows other optimizing manipulations involving loops, most notably loop condition joining, but to a lesser degree loop splitting, to occur. A conditional statement may refer to the loop control not of its immediately encompassing `for` statement, but rather to the outer `for` statement. This occurs in many cases in the image algebra, and will appear in an example discussed in Chapter 6. The loop interchange is crucial in allowing more powerful transformations to take place.

Figure 5.2 shows a nested loop before and after loop interchange, with the loops unrolled to show the exact statement ordering. While it is fairly straightforward to see the difference in the code before and after loop interchange, it is awkward to express this difference mathematically. Transition from row-major to column-major involves moving statements over large groups of statements. Figure 5.3 shows some of this movement. To transform row-major to column-major, the statement $S[n_1/x_2][m_1 + 1/x_1]$ must be interchanged with all of the statements above it except $S[n_1/x_2][m_1/x_1]$ (that is, all of the statements with a lower row number and a higher

The code segments:

(in row-major form)	(in column-major form)
for $x_1 := m_1$ **to** m_2 **do**	**for** $x_2 := n_1$ **to** n_2 **do**
for $x_2 := n_1$ **to** n_2 **do**	**for** $x_1 := m_1$ **to** m_2 **do**
S	S
od	**od**
od	**od**

are equivalent to the following statements (assuming $m_1 \leq m_2$ and $n_1 \leq n_2$):

$S[n_1/x_2][m_1/x_1];$	$S[m_1/x_1][n_1/x_2];$
$S[n_1+1/x_2][m_1/x_1];$	$S[m_1+1/x_1][n_1/x_2];$
$S[n_1+2/x_2][m_1/x_1];$	$S[m_1+2/x_1][n_1/x_2];$
$\vdots$	$\vdots$
$S[n_2/x_2][m_1/x_1];$	$S[m_2/x_1][n_1/x_2];$
$S[n_1/x_2][m_1+1/x_1];$	$S[m_1/x_1][n_1+1/x_2];$
$\vdots$	$\vdots$
$S[n_2/x_2][m_1+1/x_1];$	$S[m_2/x_1][n_1+1/x_2];$
$\vdots$	$\vdots$
$S[n_2/x_2][m_2/x_1];$	$S[m_2/x_1][n_2/x_2];$

Figure 5.2. The Effects of Loop Interchange

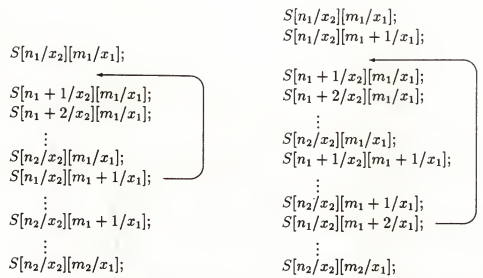

Figure 5.3. Statement Interchanging During Loop Interchanging

column number). Then, the statement $S[n_1/x_2][m_1 + 2/x_1]$ must be interchanged with all of the statements above it except the first two (again, all statements except those with a lower row number and a lower column number). A statement in row i and column j must be interchanged with statements in all previous rows which have a higher column number.

As a preliminary result to loop interchange, notice that if there is no body to a loop, the loop has no meaning.

Lemma 5.2.1 (Loop simplification)

for $x := s_1$ **to** s_2 **do** D **od** $= D$

The proof of this follows directly from loop unrolling.

With this result, the proof of the validity of loop interchange is fairly straightforward.

Theorem 5.2.2 (Loop Interchange)

$\models \mathcal{M}$(for $x_1 := s_1$ to s_2 do for $x_2 := s_3$ to s_4 do S od od)$(\sigma) =$

$\mathcal{M}$(for $x_2 := s_3$ to s_4 do for $x_1 := s_1$ to s_2 do S od od)(σ)

Provided:

sets$(S) \cap$ uses$(s_1) = \phi$

sets$(S) \cap$ uses$(s_2) = \phi$

sets$(S) \cap$ uses$(s_3) = \phi$

sets$(S) \cap$ uses$(s_4) = \phi$

$x_1 \notin ivar\ (s_3) = \phi$

$x_1 \notin ivar\ (s_4) = \phi$

$\forall y_1, y_2, y_3, y_4 \ni \mathcal{R}(s_1)(\sigma) \leq y_1 < y_2 \leq \mathcal{R}(s_2)(\sigma)$, *and* $\mathcal{R}(s_3)(\sigma) \leq y_3 < y \leq \mathcal{R}(s_4)(\sigma)$,

$S[y_3/x_2][y_2/x_1]$ *is interchangeable with* $S[y_4/x_2][y_1/x_1]$

Proof: Proof by induction on the number of times through the loop, $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1)$.

Basis: $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) \leq 0$ (no iterations).
$\mathcal{M}$ (for $x_1 := s_1$ to s_2 do for $x_2 := s_3$ to s_4 do S od od)(σ)
$= \mathcal{M}(D)(\sigma)$ (Loop unrolling [4.3.1])
$= \mathcal{M}$(for $x_2 := s_3$ to s_4 do D od)(σ) (for simplification [5.2.1])
$= \mathcal{M}$(for $x_2 := s_3$ to s_4 do for $x_1 := s_1$ to s_2 do S od od)(σ)
(Loop rolling [4.3.1])

Induction step: Assume that
for $x_1 := s_1$ to s_2 do for $x_2 := s_3$ to s_4 do S od od =
for $x_2 := s_3$ to s_4 do for $x_1 := s_1$ to s_2 do S od od
in states where $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) = k\ (k \geq 0)$.

Show it is true in states where $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) = k + 1$.
$\mathcal{M}$ (for $x_1 := s_1$ to s_2 do for $x_2 := s_3$ to s_4 do S od od)(σ)
$= \mathcal{M}$ (for $x_1 := s_1$ to $s_2 - 1$ do for $x_2 := s_3$ to s_4 do S od od;
(for $x_2 := s_3$ to s_4 do S od)$[\overline{\mathcal{R}(s_2)(\sigma)}/x_1])(\sigma)$
(Loop unrolling [4.3.2])

$= \mathcal{M}$ (**for** $x_2 := s_3$ **to** s_4 **do for** $x_1 := s_1$ **to** $s_2 - 1$ **do** S **od od**;
(**for** $x_2 := s_3$ **to** s_4 **do** S **od**)$[\overline{\mathcal{R}(s_2)(\sigma)}/x_1])(\sigma)$
(Induction hypothesis)

$= \mathcal{M}$ (**for** $x_2 := s_3$ **to** s_4 **do for** $x_1 := s_1$ **to** $s_2 - 1$ **do** S **od od**;
for $x_2 := s_3[\overline{\mathcal{R}(s_2)(\sigma)}/x_1]$ **to** $s_4[\overline{\mathcal{R}(s_2)(\sigma)}/x_1]$ **do**
$S[\overline{\mathcal{R}(s_2)(\sigma)}/x_1]$ **od**)(σ) (Def. of subst. into stat. [3.4.4])

$= \mathcal{M}$ (**for** $x_2 := s_3$ **to** s_4 **do for** $x_1 := s_1$ **to** $s_2 - 1$ **do** S **od od**;
for $x_2 := s_3$ **to** s_4 **do** $S[\overline{\mathcal{R}(s_2)(\sigma)}/x_1]$ **od od**)(σ)
(Substitution simplification [5.3.2])

$= \mathcal{M}$ (**for** $x_2 := s_3$ **to** s_4 **do for** $x_1 := s_1$ **to** $s_2 - 1$ **do** S **od**;
$S[\overline{\mathcal{R}(s_2)(\sigma)}/x_1][x_2/x_2]$ **od**) (σ) (Loop joining [5.1.1])

$= \mathcal{M}$ (**for** $x_2 := s_3$ **to** s_4 **do for** $x_1 := s_1$ **to** $s_2 - 1$ **do** S **od**;
$S[\overline{\mathcal{R}(s_2)(\sigma)}/x_1]$ **od**) (σ) (Def. of subst. [3.4.4])

$= \mathcal{M}$ (**for** $x_2 := s_3$ **to** s_4 **do for** $x_1 := s_1$ **to** s_2 **do** S **od od**) (σ)
(Loop rolling [4.3.2])

□

5.3 Code Motion

Code motion is a traditional code transformation which removes loop invariant statements from a loop. It can be viewed as a series of smaller transformations. First the statement to be removed is interchanged with the other statements in the loop until it becomes the first statement. (If this cannot be done, the statement cannot be removed.) Then the statements in the loop are unrolled. The statements which are invariant (since the loop may have been unrolled more than once, there may be more than one copy of the invariant statement) are moved to the first positions by more statement interchange, leaving the other statements in the same order as before. The remaining statements are rolled into the loop again and the invariant statement is compressed so there is only one copy of it. Since all of these operations have already been shown to yield code with the same meaning as the original code, the overall operation must also result in equivalent code.

Theorem 5.3.1 (Code Motion)

$\models \mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ do } S_1; S_2 \texttt{ od})(\sigma) = \mathcal{M}(S_1; \texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ do } S_2 \texttt{ od})(\sigma)$

Provided:

$\text{sets}(S_1) \cap \text{uses}(s_1) = \phi$

$\text{sets}(S_1) \cap \text{uses}(s_2) = \phi$

$\mathcal{R}(s_1)(\sigma) \leq \mathcal{R}(s_2)(\sigma)$;

$x \notin \mathit{ivar}(S_1)$;

S_1 *is interchangeable with* $S_2[k/x]$ *for* $k = m \ldots n-1$;

$S_1; S_1$ *is compressible.*

In proving this theorem, the following lemmas are very useful.

Lemma 5.3.2

If $x \notin \mathit{ivar}(s)$ *then* $s[s_1/x] \equiv s$. *Similarly, if* $x \notin \mathit{ivar}(b)$ *then* $b[s_1/x] \equiv b$.

Lemma 5.3.3

If $x \notin \mathit{ivar}(S)$ *then* $\mathcal{S}[s_1/x] \equiv S$.

The proofs of both of these lemmas follow by mathematical induction on the complexity of s, b, and S. The proof of code motion then follows by induction on the number of times through the loop, $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1)$.

Proof:
Basis: $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) = 1$.

$\mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ do } S_1; S_2 \texttt{ od})(\sigma)$

$= \mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 - 1 \texttt{ do } S_1; S_2 \texttt{ od};$
$(S_1; S_2)[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$ (Loop unrolling [4.3.2])

$= \mathcal{M}(\text{D};(S_1; S_2)[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$ (Loop unrolling [4.3.1])

$= \mathcal{M}((S_1; S_2)[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$ (Def. of D [3.5.3])

$= \mathcal{M}(S_1[\overline{\mathcal{R}(s_2)(\sigma)}/x]; S_2[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$
(Def. of subst. into stat. [3.4.4])

$= \mathcal{M}(S_1; S_2[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\sigma)$ (Subst. simplification [5.3.3])

$= \mathcal{M}(S_2[\overline{\mathcal{R}(s_2)(\sigma)}/x])(\mathcal{M}(S_1)(\sigma))$ (Def. of $S_1; S_2$ [3.5.3])
$= \mathcal{M}(S_2[\overline{\mathcal{R}(s_2)(\sigma)}/x]; D)(\mathcal{M}(S_1)(\sigma))$ (Def. of D [3.5.3])
$= \mathcal{M}(S_2[\overline{\mathcal{R}(s_2)(\sigma)}/x];$
`for` $x :=$ $\overline{\mathcal{R}(s_1)(\sigma)} + 1$ `to` $\overline{\mathcal{R}(s_2)(\sigma)}$ `do` S `od`)
$(\mathcal{M}(S_1)(\sigma))$ (Loop rolling [4.3.1])
$= \mathcal{M}(S_2[\overline{\mathcal{R}(s_2)(\mathcal{M}(S_1)(\sigma))}/x];$
`for` $x :=$ $\overline{\mathcal{R}(s_1)(\mathcal{M}(S_1)(\sigma))} + 1$ `to` $\overline{\mathcal{R}(s_2)(\mathcal{M}(S_1)(\sigma))}$ `do` S `od`)
$(\mathcal{M}(S_1)(\sigma))$ (Def. of empty *sets* and *uses* [3.8.3])
$= \mathcal{M}($`for` $x :=$ s_1 `to` s_2 `do` S_2 `od`$)(\mathcal{M}(S_1)(\sigma))$
(Loop rolling [4.3.1])
$= \mathcal{M}(S_1;$ `for` $x :=$ s_1 `to` s_2 `do` S_2 `od`$)(\sigma)$
(Def. of $S_1; S_2$ [3.5.3])

Induction step: Assume that
`for` $x :=$ s_1 `to` s_2 `do` $S_1; S_2$ `od` $= S_1;$ `for` $x :=$ s_1 `to` s_2 `do` S_2 `od`
in states where $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) = k$ $(k \geq 1)$.

Show it is true in states where $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) = k + 1$.
$\mathcal{M}($`for` $x :=$ s_1 `to` s_2 `do` $S_1; S_2$ `od`$)(\sigma)$
$= \mathcal{M}((S_1; S_2)[\overline{\mathcal{R}(s_1)(\sigma)}/x];$
`for` $x :=$ $\overline{\mathcal{R}(s_1)(\sigma)} + 1$ `to` $\overline{\mathcal{R}(s_2)(\sigma)}$ `do` $S_1; S_2$ `od`$)(\sigma)$
(Loop unrolling [4.3.2])
$= \mathcal{M}(S_1[\overline{\mathcal{R}(s_1)(\sigma)}/x]; S_2[\overline{\mathcal{R}(s_1)(\sigma)}/x];$
`for` $x :=$ $\overline{\mathcal{R}(s_1)(\sigma)} + 1$ `to` $\overline{\mathcal{R}(s_2)(\sigma)}$ `do` $S_1; S_2$ `od`$)(\sigma)$
(Def. of subst. into stat. [3.4.4])
$= \mathcal{M}(S_1; S_2[\overline{\mathcal{R}(s_1)(\sigma)}/x];$
`for` $x :=$ $\overline{\mathcal{R}(s_1)(\sigma)} + 1$ `to` $\overline{\mathcal{R}(s_2)(\sigma)}$ `do` $S_1; S_2$ `od`$)(\sigma)$
(Subst. simplification [5.3.3])
$= \mathcal{M}(S_1; S_2[\overline{\mathcal{R}(s_1)(\sigma)}/x];$
$S_1;$ `for` $x :=$ $\overline{\mathcal{R}(s_1)(\sigma)} + 1$ `to` $\overline{\mathcal{R}(s_2)(\sigma)}$ `do` S_2 `od`$)(\sigma)$
(Induction hypothesis)
$= \mathcal{M}(S_1; S_1; S_2[\overline{\mathcal{R}(s_1)(\sigma)}/x];$
`for` $x :=$ $\overline{\mathcal{R}(s_1)(\sigma)} + 1$ `to` $\overline{\mathcal{R}(s_2)(\sigma)}$ `do` S_2 `od`$)(\sigma)$
(Statement interchange [4.1.1])
$= \mathcal{M}(S_1; S_2[\overline{\mathcal{R}(s_1)(\sigma)}/x];$
`for` $x :=$ $\overline{\mathcal{R}(s_1)(\sigma)} + 1$ `to` $\overline{\mathcal{R}(s_2)(\sigma)}$ `do` S_2 `od`$)(\sigma)$
(Statement compression [4.1.6])
$= \mathcal{M}(S_2[\overline{\mathcal{R}(s_1)(\sigma)}/x];$
`for` $x :=$ $\overline{\mathcal{R}(s_1)(\sigma)} + 1$ `to` $\overline{\mathcal{R}(s_2)(\sigma)}$ `do` S_2 `od`$)(\mathcal{M}(S_1)(\sigma))$
(Def. of $S_1; S_2$ [3.5.3])

$= \mathcal{M}(S_2[\overline{\mathcal{R}(s_1)(\mathcal{M}(S_1)(\sigma))}/x];$
  `for` $x := \overline{\mathcal{R}(s_1)(\mathcal{M}(S_1)(\sigma))} + 1$
    `to` $\overline{\mathcal{R}(s_2)(\mathcal{M}(S_1)(\sigma))}$ `do` S_2 `od`$)(\mathcal{M}(S_1)(\sigma))$
(Def. of empty *sets* and *uses* [3.8.3])
$= \mathcal{M}($`for` $x := s_1$ `to` s_2 `do` S_2 `od`$)(\mathcal{M}(S_1)(\sigma))$
(Loop rolling [4.3.2])
$= \mathcal{M}(S_1;$ `for` $x := s_1$ `to` s_2 `do` S_2 `od`$)(\sigma)$
(Def. of $S_1; S_2$ [3.5.3])

□

5.4 Loop-Conditional Joining

Most of the conventional optimizations, while they may increase the speed of code significantly, do not actually change the order of complexity of the code. If an algorithm is $O(n^2)$ before the optimization, it is probably still $O(n^2)$ after the optimization. (There are of course some cases where an optimization may completely eliminate a block of code, effectively reducing its running time to $O(1)$.) Loop-conditional joining may change the number of times a loop is executed and in some cases may reduce the number of times a loop is executed to a constant such as one or two.

Loop-conditional joining is applied to a loop where the only statement in the loop is an **if then** statement which compares the loop control variable with an expression which does not depend on either the loop control variable or statement in the loop for a value. If the conditional has an **else** clause, the **if then else** statement must first be split (using Theorem 4.2.3), and the loop split into two loops containing one **if then** statement each (using Theorem 5.1.1). The loop is unrolled (using Theorems 4.3.1 and 4.3.2) and the conditional is simplified (using Theorem 4.2.5). This will give a group of zero or more statements which no longer have the original conditional, which are then rerolled into another loop (or left as if there are only a few statements).

Theorem 5.4.1 (Loop-Conditional Joining)

$\models$ `for` $x := s_1$ `to` s_2 `do if` $x \geq s_3$ `then` S `fi od` = `for` $x := max(s_1, s_3)$ `to` s_2 `do` S `od`

and

$\models$ `for` $x := s_1$ `to` s_2 `do if` $x \leq s_3$ `then` S `fi od` = `for` $x := s_1$ `to` $min(s_2, s_3)$ `do` S `od`

Provided:

sets$(S) \cap$ uses$(s_3) = \phi$

$x \notin$ *uses*(s_3)

Proof: Proof of the first part by induction on the number of times through the loop, $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1)$. The proof of the second part is nearly identical.

Basis: $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) \leq 0$ (no iterations).
$\mathcal{M}$(`for` $x := s_1$ `to` s_2 `do if` $x \geq s_3$ `then` S `fi od`)(σ)
$= \mathcal{M}(\mathrm{D})(\sigma)$ (Loop unrolling [4.3.1])
$= \mathcal{M}$(`for` $x :=$ max (s_1, s_3) `to` s_2 `do if` $x \geq s_3$ `then` S `fi od`)(σ)
(Loop rolling [4.3.1])

Induction step: Assume that
`for` $x := s_1$ `to` s_2 `do if` $x \geq s_3$ `then` S `fi od` = `for` $x :=$ max(s_1, s_3) `to` s_2 `do` S `od`
in states where $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) = \mathrm{k}$ $(\mathrm{k} \geq 0)$.

Show it is true in states where $(\mathcal{R}(s_2)(\sigma) - \mathcal{R}(s_1)(\sigma) + 1) = \mathrm{k} + 1$.
$\mathcal{M}$(`for` $x := s_1$ `to` s_2 `do if` $x \geq s_3$ `then` S `fi od`)(σ)
$= \mathcal{M}$((`if` $x \geq s_3$ `then` S `fi`)$[\overline{\mathcal{R}(s_1)(\sigma)}/x]$;
`for` $x := \overline{\mathcal{R}(s_1)(\sigma)} + 1$ `to` $\overline{\mathcal{R}(s_2)(\sigma)}$ `do`
`if` $x \geq s_3$ `then` S `fi od`)(σ) (Loop unrolling [4.3.2])
$= \mathcal{M}$(`if` $x[\overline{\mathcal{R}(s_1)(\sigma)}/x] \geq s_3[\overline{\mathcal{R}(s_1)(\sigma)}/x]$ `then` $S[\overline{\mathcal{R}(s_1)(\sigma)}/x]$ `fi`;
`for` $x := \overline{\mathcal{R}(s_1)(\sigma)} + 1$ `to` $\overline{\mathcal{R}(s_2)(\sigma)}$ `do`
`if` $x \geq s_3$ `then` S `fi od`)(σ) (Def. of subst. into stat.[3.4.4])
$= \mathcal{M}$(`if` $\overline{\mathcal{R}(s_1)(\sigma)} \geq s_3$ `then` $S[\overline{\mathcal{R}(s_1)(\sigma)}/x]$ `fi`;
`for` $x := \overline{\mathcal{R}(s_1)(\sigma)} + 1$ `to` $\overline{\mathcal{R}(s_2)(\sigma)}$ `do`
`if` $x \geq s_3$ `then` S `fi od`)(σ) (Def. of subst. into exp. [3.4.1])

$= \mathcal{M}($**if** $\overline{\mathcal{R}(s_1)(\sigma)} \geq s_3$ **then** $S[\overline{\mathcal{R}(s_1)(\sigma)}/x]$ **fi**;
for $x :=$ max $(\overline{\mathcal{R}(s_1)(\sigma)} + 1, s_3)$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)(\sigma)$
(Induction hypothesis)
$= \mathcal{M}($**for** $x :=$ max $(\overline{\mathcal{R}(s_1)(\sigma)} + 1, s_3)$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)$
$(\mathcal{M}($**if** $\overline{\mathcal{R}(s_1)(\sigma)} \geq s_3$ **then** $S[\overline{\mathcal{R}(s_1)(\sigma)}/x]$ **fi**$)(\sigma))$
(Def. of $S_1; S_2$ [3.5.3])
$= \mathcal{M}($**for** $x :=$ max $(\overline{\mathcal{R}(s_1)(\sigma)} + 1, s_3)$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)$
$(\mathit{if}\ \mathcal{R}(\overline{\mathcal{R}(s_1)(\sigma)})(\sigma) \geq \mathcal{R}(s_3)(\sigma)\ \mathit{then}\ \mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma)$
$\mathit{else}\ \mathcal{M}(D)(\sigma)\ \mathit{fi})$ (Def. of **if** [3.5.3])
$= \mathcal{M}($**for** $x :=$ max $(\overline{\mathcal{R}(s_1)(\sigma)} + 1, s_3)$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)$
$(\mathit{if}\ \overline{\mathcal{R}(s_1)(\sigma)} \geq \mathcal{R}(s_3)(\sigma)\ \mathit{then}\ \mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma)$
$\mathit{else}\ \mathcal{M}(D)(\sigma)\ \mathit{fi})$ (Sem. of $\mathcal{R}(m)$ [3.5.1])
$= \mathit{if}\ \overline{\mathcal{R}(s_1)(\sigma)} \geq \mathcal{R}(s_3)(\sigma)\ \mathit{then}$
$\mathcal{M}($**for** $x :=$ max $(\overline{\mathcal{R}(s_1)(\sigma)} + 1, s_3)$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)$
$(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$
$\mathit{else}\ \mathcal{M}($**for** $x :=$ max $(\overline{\mathcal{R}(s_1)(\sigma)} + 1, s_3)$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)$
$(\mathcal{M}(D)(\sigma))\ \mathit{fi}$ (Meaning of if)
$= \mathit{if}\ \overline{\mathcal{R}(s_1)(\sigma)} \geq \mathcal{R}(s_3)(\sigma)\ \mathit{then}$
$\mathcal{M}($**for** $x :=$ max $(\overline{\mathcal{R}(s_1)(\sigma)} + 1, s_3)$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)$
$(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$
$\mathit{else}\ \mathcal{M}($**for** $x :=$ max $(\overline{\mathcal{R}(s_1)(\sigma)} + 1, s_3)$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)(\sigma)$
fi (Meaning of D [3.5.3])
$= \mathit{if}\ \overline{\mathcal{R}(s_1)(\sigma)} \geq \mathcal{R}(s_3)(\sigma)\ \mathit{then}$
$\mathcal{M}($**for** $x :=$ max $(\overline{\mathcal{R}(s_1)(\sigma)} + 1, s_3)$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)$
$(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$
$\mathit{else}\ \mathcal{M}($**for** $x :=$ max $(s_1 + 1, s_3)$ **to** s_2 **do** S **od**$)(\sigma)\ \mathit{fi}$
(Sem. of $\mathcal{R}(s)$ [3.5.1])
$= \mathit{if}\ \overline{\mathcal{R}(s_1)(\sigma)} \geq \mathcal{R}(s_3)(\sigma)\ \mathit{then}$
$\mathcal{M}($**for** $x :=$ max $(\overline{\mathcal{R}(s_1)(\sigma)} + 1, s_3)$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)$
$(\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x])(\sigma))$
$\mathit{else}\ \mathcal{M}($**for** $x :=$ max (s_1, s_3) **to** s_2 **do** S **od**$)\ (\sigma)\ \mathit{fi}$
(Since $\mathcal{R}(s_1)(\sigma) < \mathcal{R}(s_3)(\sigma)$,
max (s_1, s_3) = max $(s_1 + 1, s_3)$)
$= \mathit{if}\ \overline{\mathcal{R}(s_1)(\sigma)} \geq \mathcal{R}(s_3)(\sigma)\ \mathit{then}$
$\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x];$
for $x :=$ max $(\overline{\mathcal{R}(s_1)(\sigma)} + 1, s_3)$ **to** $\overline{\mathcal{R}(s_2)(\sigma)}$ **do** S **od**$)(\sigma)$
$\mathit{else}\ \mathcal{M}($**for** $x :=$ max (s_1, s_3) **to** s_2 **do** S **od**$)\ (\sigma)\ \mathit{fi}$
(Def. of $S_1; S_2$ [3.5.3])
$= \mathit{if}\ \overline{\mathcal{R}(s_1)(\sigma)} \geq \mathcal{R}(s_3)(\sigma)\ \mathit{then}$
$\mathcal{M}(S[\overline{\mathcal{R}(s_1)(\sigma)}/x];$

$$\begin{aligned}
&\texttt{for } x := \overline{\mathcal{R}(s_1)(\sigma)} + 1 \texttt{ to } \overline{\mathcal{R}(s_2)(\sigma)} \texttt{ do } S \texttt{ od})(\sigma) \\
&\quad \textit{else } \mathcal{M}(\texttt{for } x := \max\,(s_1,\, s_3) \texttt{ to } s_2 \texttt{ do } S \texttt{ od})\,(\sigma) \textit{ fi} \\
&\qquad\qquad (\overline{\mathcal{R}(s_1)(\sigma)} \geq \mathcal{R}(s_3)(\sigma)) \\
= \; &\textit{if } \overline{\mathcal{R}(s_1)(\sigma)} \geq \mathcal{R}(s_3)(\sigma) \textit{ then} \\
&\quad \mathcal{M}(\texttt{for } x := s_1 \texttt{ to } s_2 \texttt{ do } S \texttt{ od})(\sigma) \\
&\quad \textit{else } \mathcal{M}(\texttt{for } x := \max\,(s_1,\, s_3) \texttt{ to } s_2 \texttt{ do } S \texttt{ od})\,(\sigma) \textit{ fi} \\
&\qquad\qquad \text{(Loop rolling [4.3.2])} \\
= \; &\textit{if } \overline{\mathcal{R}(s_1)(\sigma)} \geq \mathcal{R}(s_3)(\sigma) \textit{ then} \\
&\quad \mathcal{M}(\texttt{for } x := \max\,(s_1,s_3) \texttt{ to } s_2 \texttt{ do } S \texttt{ od})(\sigma) \\
&\quad \textit{else } \mathcal{M}(\texttt{for } x := \max\,(s_1,\, s_3) \texttt{ to } s_2 \texttt{ do } S \texttt{ od})\,(\sigma) \textit{ fi} \\
&\qquad\qquad (\overline{\mathcal{R}(s_1)(\sigma)} \geq \mathcal{R}(s_3)(\sigma)) \\
= \; &\mathcal{M}(\texttt{for } x := \max\,(s_1,s_3) \texttt{ to } s_2 \texttt{ do } S \texttt{ od})(\sigma) \\
&\qquad\qquad \text{(Meaning of if)}
\end{aligned}$$

□

CHAPTER 6
A PROTOTYPE OPTIMIZER

The previous chapters provide the necessary denotational semantic background for performing code optimizations. This background certainly provides the provable correctness of the transformations and related optimizations. Still, it remains to be shown that these transformations can be implemented in a reasonable fashion.

This research has revealed that they can indeed be implemented to give the basic tools for a person to use to perform a variety of potential optimizations and evaluate the results. This chapter will describe the Heursitic Optimizing Prototype System (HOPS), a LISP system that provides all of the primitive and global optimizations discussed in Chapters 4 and 5. This chapter gives a brief description of HOPS, starting with an overview of the system in Section 6.1. In developing the system the need for a timer for the code and a better approximation of when two sets of variables are always separate became apparent. The timer which resulted is discussed in Section 6.2 and the refinement of static approximation for always separate is presented in Section 6.3. Once the basic transformations were implemented, it was necessary to write programs to combine them in an attempt to optimize code. A discussion of these heuristic programs is in Section 6.4. Finally, a large example of HOPS in action, optimizing a program to do image histograms, is presented in Section 6.5.

6.1 The System and Its Data Structures

HOPS is an interactive LISP system, implemented in XLISP running under UNIX, on a Sun 3/280 and a Gould Powernode 9080. It has functions that perform each of

the primitive transformations as described in Chapter 4 and each of the optimizing transformations as described in Chapter 5. It also provides functions to aid a human optimizer, including statement and expression construction, extraction of parts of statements and expressions, statement and expression validity checks, peephole optimizations, and, probably most importantly, statement timing. The timer is discussed in more detail in Section 6.2 and the other functions are described later in this section.

Simple variables and constants in this language are LISP atoms and indexed variables, expressions and statements are lists. Variables must be declared in HOPS programs, unlike programs in the language described in Chapter 3. This can be done with the function `MakeVariable`. Statements and expressions are stored in prefix form. The format for storing each of these is given in Table 6.1.

In order to simplify some testing, abstract statements and expressions are allowed. Thus, if the user is interested in examining `if` statement transformations, the user need not enter complete statements for each clause, but may instead enter abstract statements for these unimportant clauses. For example, a user interested in loop-conditional joining may not want to consider the statement in the `then` clause of the conditional being joined. In this case, an abstract statement could be used. Similarly, someone trying to explore `if` statement simplification as described in Theorem 4.2.5 could use an abstract boolean expression rather than some real expression. Abstract values are assumed to set and use no variables. These abstract values are atoms and must be declared by the user using the functions `MakeAbstractStatement`, `MakeAbstractBoolean`, and `MakeAbstractExpression`.

The functions in HOPS each take a single statement as a parameter and return the statement after execution of the transformation, if the transformation is valid.

Table 6.1. HOPS Equivalents for Language Constructs

Language Construct	HOPS Equivalent	
Integer Variables		
`x`	`x`	
`a[s]`	`(a s)`	
Integer Expressions		
`m`	`m`	
`s1 op s2`	`(op s1 s2)`	legal ops are `+, -, *, /`
`if b then s1 else s2 fi`	`(if b s1 s2)`	
Boolean Expressions		
`true`	`true`	
`false`	`false`	
`s1 op s2`	`(op s1 s2)`	legal ops are `<, >, <=, >=, =, <>`
`¬b`	`(not b)`	
`s in s1...s2`	`(in s (s1 s2))`	
Statements		
`v := s`	`(:= v s)`	
`S1; S2`	`(S1 S2)`	
`if b then S1 else S2 fi`	`(if b S1 S2)`	
`D`	`()`	
`for x := s1 to s2 do S od`	`(for x s1 s2 S)`	

There are a number of functions allowing the user to extract single statements from structured statements, making it easier to send the single statement expected here as a parameter. These are relatively straightforward and are omitted from the present discussion. The important transforming functions include the following.

Interchange. This function will interchange the first two statements in a compound statement.

InterchangeWithSubstitution. This function interchanges the first two statements of a compound list. The first statement must be an assignment statement and the value assigned to the left-hand-side is substituted for the left-hand-side in the second statement.

InterchangeWithBackSubstitution. This function performs interchange with backward substitution as described in Theorem 4.1.5.

LoopUnroll. This function unrolls a statement from the back of a `for` statement. Currently, this is only done if the loop bounds are constants—there is no attempt made to evaluate variable loop bounds.

AbsorbIn. This function will attempt to absorb statements into an `if` statement from either before or after the `if` statement. If only absorption of statements following the `if` statement is desired, `AbsorbIn1` can be used, while `AbsorbIn2` only attempts to absorb statements from before the `if` statement.

ExtractOut. This function removes statements from both branches of an `if` statement to either before or after the statement. `ExtractOut1` and `ExtractOut2` will move statements to after or before the `if` statement respectively.

AbsorbWithSubstitution. This function absorbs assignment statements before `if` statements into the `if` statement, substituting the right-hand-side for the left hand side in the `if` statement condition as necessary.

IfSplit. This function divides an `if then else` statement into two `if then` statements.

LoopInterchange. This function will interchange the boundaries of nested loops.

LoopJoin. This function will join a pair of `for` loops with the same range.

LoopSplit. This function will split a single `for` loop into a pair of `for` loops with the same range.

MoveCode. This function will remove loop invariant statements to before `for` loops.

LCJoin. This function will convert a loop statement with a nested conditional statement into a loop statement with altered bounds.

StatementSimplify. This function performs a variety of peephole optimizations.

Each of the functions of HOPS is nondestructive. This enables a user to assign the value of a statement to a variable and then to try a variety of transformations on the variable, being sure of always starting with the same statement. A typical call sequence might then be:

```
(setq testprog '(for x 1 10 ( (:= y 1) (:= x (+ y x)) ) ) )
                        Give the test program a value.
(time testprog)         Compute the original time it takes.
(setq try1 (MoveCode testprog))
                        Attempt code motion.
(time try1)             Compute the time the transformed program takes.
(setq try2 (LCJoin testprog))
                        Attempt loop-conditional joining.
(time try2)             Compute the time the transformed program takes.
```

Notice that the user tried to perform loop-conditional joining when it was not possible (since there is no conditional). HOPS recognizes this and will not alter the statement, so the time for the original statement and the statement after the loop-conditional joining will be the same.

6.2 A Parameterized Timer

The functions given in the previous section provide the user with the ability to attempt a variety of transformations, but there is nothing in HOPS which will tell the user when one transformation provides improved code. Instead, HOPS provides a pair of timers, one which returns a symbolic time and the other which simply returns a count of time units, which the user can then use to determine the benefit (or harm) of the transformation.

Since many systems differ in the amount of time it may take to perform operations, these timers are based on a set of constants, any of which may be changed by the user to better represent relative speeds of the user's actual system. An optimizing system based on HOPS could then use the symbolic time to determine whether or not to apply a particular transformation. Additionally, there are two functions to return the time of boolean and integer operators, so that different operators may be assigned different times (as is so often the case in actual systems). When the bounds of a loop can be statically evaluated, the actual number of iterations in the loop will be computed, but if the loop bounds are expressions, another default will give the number of times through the loop. Figure 6.2 shows the symbolic times of both the original and altered statements in the example at the end of the previous section.

6.3 A New Approximation of Always Separate

Using ivar and livar as the static approximation of *sets* and *uses* in Section 3.9 proved inadequate in HOPS. While the approximation was fine for simple variables,

Original statement: `(for x 1 10 ( (:= y 1) (:= x (+ y x)) ) ) )`
Its symbolic time:

```
(+   ForStatementTime
     (*   (+   1 (- 10 1))
          (+   CompoundStatementTime
               (+   (+   AssignmentStatementTime
                         ConstantTime
                    )
                    (+   AssignmentStatementTime
                         (+   BinaryFunctionTime
                              VariableTime
                              VariableTime
                         )
                    )
               )
          )
     )
)
```

Statement transformed by MoveCode:
`(( := y 1) (for x 1 10 ( (:= x (+ y x)) ) ) )`
Its symbolic time:

```
(+   CompoundStatementTime
     (+   AssignmentStatementTime
          ConstantTime
     )
     (+   ForStatementTime
          (*   (+   1 (- 10 1))
               (+   BinaryFunctionTime
                    VariableTime
                    VariableTime
               )
          )
     )
)
```

Figure 6.1. Some Symbolic Statement Times

it was far too broad to say that the entire array was set when in fact only one element of it may have been set. Too many image processing functions work on arrays, usually going through them in some specific order each time. Because of this, a new representation of intermediate variables was used in HOPS.

Simple variables are still stored as simple variables in the intermediate form. Array variables are now stored as both the array and some index information. This array index information is stored in one of five formats, depending on how the array reference appears in the program. These index types are described below.

Constants. If the array is indexed by a constant or constant expression (and thus the reference is of the form $a[m]$), the index will be stored as the constant m.

Variables. If the array is indexed by a variable (and thus the reference is of the form $a[x]$), the index will be stored as the variable x.

Expressions. If the array is indexed by a variable expression (and thus the reference is of the form $a[s]$), the index will be stored as the expression s (where expressions are stored as discussed in Section 6.1).

Constant subranges. If the array reference is in a `for` loop with constant bounds and is indexed by the loop control variable, the index will be stored as a subrange of the constants in the form `(lower-bound upper-bound)`.

Variable subranges. If the array reference is in a `for` loop with nonconstant bounds or is indexed by a function of the loop control variable, the index will be stored as a subrange of the expressions in the form `(lower-bound-expression upper-bound-expression)`.

As before, any two simple variables are always separate if they have different names and any simple variable is always separate from any array variable. When two intermediate variables are array references referring to the same array, it is necessary to check the index information. In some cases, it may be possible to determine from

Table 6.2. When Two Index Types May Be Always Separate

Type of second index	Type of first index: Constant	Variable	Expressions	Constant Subrange	Variable Subrange
Constant	Maybe	Never	Never	Maybe	Never
Variable	Never	Never	Maybe	Never	Never
Expression	Never	Maybe	Maybe	Never	Never
Constant Subrange	Maybe	Never	Never	Maybe	Never
Variable Subrange	Never	Never	Never	Never	Maybe

the indices that the array references are always separate. Table 6.2 tells in which cases, based on the index values, array references may be always separate.

In the cases in Table 6.2 marked *Maybe*, HOPS will check the values of the two indices to determine if they can positively be declared to be always separate. If, for example, the first index were a constant and the second were a constant subrange, HOPS would only have to check if the constant fell into the subrange. If it did not, the two array references could safely be declared always separate. The spaces marked *Never* do not mean that there is no way for the references to be always separate, only that HOPS cannot determine statically if they were. Certainly, two arrays referenced by variables may be references to different locations, but with no information about the values of the indexing expressions, HOPS cannot conclude this.

6.4 Some Heuristic Programs Using the System

A user could certainly work with these transformations to transform code in their raw form, but they are at the level of assembly language programming. In order to assist the user, some functions have been added to attempt larger scale transformations. Thus HOPS contains some heuristics for code optimizations. These functions are described in this section.

Probably the simplest of these heuristics is `DoAllProp`, which will perform forward copy propagation wherever possible. Forward copy propagation is done by interchanging with substitution all statements of the form $x := v$ or $x := m$, to move them as far backward in the code as possible. It may also involve absorption into `if` statements. The function `PropagateAndSimplify` combines this with `StatementSimplify` to perform both peephole simplifications and forward copy statement propagation concurrently. Backward copy propagation, possible as a result of Interchange with Backward Substitution (Lemma 4.1.5), can be done with the function `BackPropagate`. Only assignment statements with an array reference on the left-hand-side will be propagated backwards.

Statement compression helps `RemoveDeadVars` with the removal of dead statements, those whose results are no longer needed by the rest of the program. Statement compression alone does not provide dead variable elimination. In order to determine which variables are live at any point in a program, there must be some notion of the output variables of that program. All previous results have considered states to be equal if they agree in the values assigned to all variables, not just a group of output variables. It is conceivable to discuss equality of states restricted to a set of variables, but is outside the scope of this research.

There are also functions to optimize the different structures, such as `CompoundOptimize` and `ForOptimize`. It is here that various ways of combining the primitive and global transformations can be explored. I conjecture that the problem of determining exactly which set of transformations will best improve the running time of any piece of code for any set of timer parameters is most certainly undecidable. However, work has been done to devise some general rules for applying the traditional global data flow transformations [6]. This work has been adapted for use by HOPS.

```
(def ForOptimize
   (lambda (Stat LiveVars)
      (prog (InnerStat NewStat SecondStat)
;  Simplify the entire statement and eliminate it if possible.
         (setq Stat
            (StatementSimplify Stat))
         (cond ( (not (IsFor Stat))
                  (return Stat))
         )
;  Then, do Backward propagation (since CompoundOptimize won't
;  know the compound is nested in an IF)
         (setq InnerStat (GetForStat Stat))
         (setq InnerStat
               (BackPropagate InnerStat (GetLCV Stat)))
;  Now optimize the inner statements
         (setq InnerStat
            (CompoundOptimize InnerStat LiveVars))
;  Restructure the statement
         (setq NewStat
               (MakeFor
                  (GetLCV Stat)
                  (GetLowBound Stat)
                  (GetHighBound Stat)
                  InnerStat
               ))
;  Attempt code motion from the loop
         (setq NewStat
                  (ForceMoveCode NewStat))
;  Attempt loop-conditional joining
         (setq NewStat
                  (ForceLCJoin NewStat))
         (return (StatementSimplify NewStat))
      )
))
```

Figure 6.2. A HOPS Program to Optimize the Histogram Program

A version of `ForOptimize` is in Figure 6.2. It begins by simplifying the entire statement with peephole optimizations. If this results in a statement which is not a loop (either because the statement in the `for` loop is nullable, or because the loop boundaries are computable and the lower bound is greater than or equal to the upper bound), the optimization of the `for` loop stops there. This initial step is time consuming and may not be needed for some `for` statements, but was determined to be necessary in the simplification of the histogram program in the next section. Next, before doing global transformations to the nested statement, backward copy propagation is done. Only statements with array references, referenced by the loop control variable are propagated backwards. Then, `CompoundOptimize` is employed to perform global transformations, such as elimination of dead variables and statement compression, on the nested statement. Once the nested statement is improved, code motion and loop-conditional joining are attempted. Finally, peephole simplifications are repeated. This entire plan of attack may constitute overkill for some `for` statements, but usually does provide improved code whenever improvements are possible. It is still up to the user to look at the times of the code with and without the improvements to determine if the improvements were actually beneficial.

There are also a variety of functions provided to extract statements of interest and greater potential for optimizations from a compound statement, along with the simplifying procedures available with HOPS. The first of these is `ExtractForStat` which will find the first `for` statement in a compound statement. Compound statements are of special interest because the amount of looping in image processing is so great. `SplitCompoundAroundIf` will extract an `if` statement which uses a particular variable (passed as a parameter) in its condition. This is particularly useful in loop-conditional joining, and there is a function, `ForceLCJoin`, which uses this to attempt a variety of changes in the code to ultimately perform loop-conditional joining.

```
(for i LowGray HighGray
     (
         (:= s 0)
         (for j LowPixel HighPixel
              (if (= i (a j))
                  (:= s (+ s 1))
                  (:= s (+ s 0))
              )
          )
          (:= (h i) s)
     )
)
```

Figure 6.3. A Straightforward Implementation of the Histogram

While none of these heuristics is altogether complicated (and they all leave it to the user to determine if indeed the transformation is beneficial), they do as a group show how extensible the basic functions of HOPS are and indicate some of its potential to become a truly intelligent code optimizer.

6.5 A Large Example: The Histogram

As an example of the abilities of HOPS, even in its present form, consider a program to compute the histogram of an image. Determining the histogram of gray levels in an image is important for a variety of techniques such as enhancement by histogram equalization over a wider range of gray levels and image segmentation [13]. To determine the histogram h of an image **a** over the gray levels `min-gray-level` to `max-gray-level` in the image algebra, the following expression is used:

`for i in min-gray-level to max-gray-level do`

$$h_i \leftarrow \Sigma(\chi_i(a))$$

`end for`

```
(for i 0 255
     (:= (h i) 0)
)
(for j 0 4095
     (for i (max 0 (a j)) (min 255 (a j))
          (:= (h i)
               (+ (h i) 1)
          )
     )
)
```

Figure 6.4. The Resulting Histogram Program

A straightforward implementation of this algorithm is shown in Figure 6.3. As stated it is extremely inefficient. It requires extra space to hold each of the new characteristic function images. It also requires looping over the size of the original image 2n times (where n is the number of gray levels)—once to compute the characteristic function and once to sum it. While standard optimizations may improve this code, they cannot eliminate the amount of looping.

The HOPS program **ForOptimize**, discussed in the previous section and presented in Figure 6.2, can be used to optimize this histogram program. The resulting code is in Figure 6.4. (Since **LowGray**, **HighGray**, **LowPixel**, and **HighPixel** are all constants, they were replaced by their corresponding values during the simplification.)

HOPS has demonstrated the potential to be an important assistant to human code optimizers. It still is fairly primitive and relies strongly on the user, both to direct its search and to determine when a transformation is beneficial. It has not yet been incorporated into any of the image algebra programming languages, and this would seem to be a logical next step. Still, its success at this level indicates the potential of approaching global optimization as a collection of primitive transformations.

CHAPTER 7
CONCLUSIONS

This dissertation presents a new approach to global optimizations. Rather than collect a variety of information about each statement and perform large transformations when the conditions are correct, I collect a small amount of information (only the *sets* and *uses* for the statement) and perform small transformations, which can then be combined to form large optimizations. By performing optimization in smaller pieces, it is easier to show that each of the pieces is correct. I have described a small language and have given its denotational definition. With this definition, I have proven that each of the primitive transformations preserves the meaning of the statement.

These primitive transformations can be combined to give global optimizations. These optimizations include some of the traditional global data flow optimizations such as code motion and copy propagation, along with some previously unexploited optimizations such as loop-conditional joining and backward copy propagation.

The Heuristic Optimizing Prototype System implements all of these primitive transformations and global optimizations. It allows a user to experiment with a variety of optimizing strategies. It also has functions developed to assist the user. These functions will attempt to rearrange the program so that some of the more beneficial optimizations (such as code motion and loop-conditional joining) can occur. The HOPS timer is configurable, enabling the user to adjust timer parameters to best represent the system being optimized.

There are three areas in which this dissertation would most logically be extended. The first is in the design of the language for the proofs. As it is now, the language is not Turing equivalent. While this would be desirable, it is not necessary for image algebra programs. It would also have greatly complicated the proofs presented here. The language could be made to have the power of a Turing machine by adding either loops which were not bounded at the time of entrance to the loop or by adding subprograms. If unbounded loops were introduced, the possibility of errors could no longer be pushed aside because infinite loops are a real possibility. Subprograms would need to have some sort of a parameter passing mechanism defined. Side effects of subprograms may increase the amount of aliasing in the language as well. These extensions, although powerful, would extend this project beyond what is necessary for image processing and would greatly increase the complexity of the proofs given.

The second area where this project could be extended is in the heuristics for applying the transformations and provided with the HOPS system. Determining when transformations that seem to degrade a piece of code might actually leave it in a position to be improved greatly is a fascinating problem, albeit one outside the scope of this dissertation. This appears on the surface to be a classical application for expert systems. It would be interesting to improve the HOPS system itself so that it could actually be used for working code optimization, or alternatively, as a teaching tool for students. This would involve some work on the interface, additional heuristic programs, and improvements to the timer such as recognizing when loops are vectorizable and permitting different guesses for the default number of loop iterations for different loops.

Finally, optimization in architectures other than traditional von Neumann architectures is a rapidly growing field. There are quite a few special architectures available for image processing [11]. Many of the optimizations applied here are also

applicable to intermediate code for multiprocessor machines such as the Connection Machine [15]. Copy propagation, loop interchange, loop joining, and loop splitting are all optimization techniques applicable to vector or concurrent computers discussed by Padua and Wolfe [21].

REFERENCES

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[2] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, July 1970.

[3] Frances E. Allen and John Cocke. A catalogue of optimizing transformations. In Randall Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[4] Dana H. Ballard and Christopher M. Brown. *Computer Vision.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[5] William A. Barrett, Rodney M. Bates, David A. Gustafson, and John D. Couch. *Compiler Construction: Theory and Practice.* Science Research Associates, Chicago, second edition, 1986.

[6] F. Chow. *A Portable Machine-Independent Global Optimizer.* PhD thesis, Stanford University, Computer Science Laboratory, Stanford, CA, 1983.

[7] John Cocke. Global common subexpression elimination. *ACM SIGPLAN Notices*, 5(7):20–25, July 1970.

[8] Patrick Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[9] Jaco de Bakker. *Mathematical Theory of Program Correctness.* Prentice-Hall, Englewood Cliffs, NJ, 1980.

[10] Veronique Donzeau-Gouge. Denotational definition of properties of program computations. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 11, pages 343–379. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[11] M.J.B. Duff and S. Levialdi, editors. *Languages and Architectures for Image Processing.* Academic Press, New York, 1981.

[12] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts.* John Wiley and Sons, Inc., New York, second edition, 1987.

[13] Rafael C. Gonzalez and Paul Wintz. *Digital Image Processing.* Addison-Wesley, Reading, MA, second edition, May 1987.

[14] Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat, editors. *Building Expert Systems.* Addison-Wesley, Reading, MA, 1983.

[15] W. Daniel Hillis. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985.

[16] Ken Kennedy. A survey of data flow analysis techniques. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5–54. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[17] Edward S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, January 1969.

[18] W.A. Martin and R.J. Fateman. The MACSYMA system. In *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 59–75, Los Angeles, 1971.

[19] S.S. Muchnick and N.D. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[20] P. Nye. S-1 U-Code:an intermediate language for Fortran and Pascal. Project document pail-8, Computer System Lab, Stanford University, Stanford, CA, October 1981.

[21] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[22] Kurt Perry. IAC: Image Algebra C. Master's thesis, University of Florida, Gainesville, FL, August 1987.

[23] G.X. Ritter and J.N. Wilson. The Image Algebra in a nutshell. In *Proceedings of the First International Conference on Computer Vision*, pages 641–645, London, England, June 1987.

[24] G.X. Ritter, J.N. Wilson, and J.L. Davidson. Image Algebra: An overview. Technical Report TR-88-05, Center for Computer Vision Research, University of Florida, Gainesville, FL, 1988.

[25] B.K. Rosen. High level data flow analysis. *Communications of the ACM*, 20(10):712–724, October 1977.

[26] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[27] Paul B. Schneck. Movement of implicit parallel and vector expressions out of program loops. *ACM SIGPLAN Notices*, 10(3):103–106, March 1975.

[28] M. Shaefer. *A Mathematical Theory of Global Program Optimization.* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[29] T.A. Standish, D.C. Harriman, D.F. Kibler, and J.M. Neighbors. The Irvine program transformation catalog. Technical Report 161, University of California at Irvine, Department of Information and Computer Science, January 1976.

[30] M.V. Zelkowitz and W.G. Bail. Optimization of structured programs. *Software—Practice and Experience*, 4(1):51–57, January 1974.

[31] Mary E. Zosel. A modest proposal for vector extensions to ALGOL. *ACM SIGPLAN Notices*, 10(3):62–71, March 1975.

BIOGRAPHICAL SKETCH

Ms. White received the degree of BA in economics from the University of Virginia in 1979 and MS in computer and information sciences from the University of Florida in 1985. She served in the Transportation Corps of the United States Army for three years and was stationed at Camp Casey, Korea, and several locations in the United States.

While at the University of Florida, Ms. White received several awards for excellence in teaching. After graduation she plans to work for Armstrong State College in Savannah, Georgia as an assistant professor in the Department of Mathematics and Computer Science. She is married to Charles Engelke.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Gerhard X. Ritter, Chairman
Professor of
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Joseph N. Wilson, Cochairman
Assistant Professor of
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Yuan-Chieh Chow
Professor of
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Manuel Bermudez
Assistant Professor of
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Richard Newman-Wolfe
Assistant Professor of
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Bruce H. Edwards
Associate Professor of
Mathematics

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

May 1990

for Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School